Torbjörn Lager

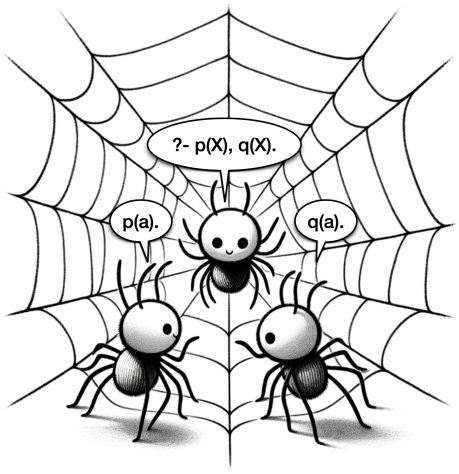
# The Prolog Trinity ecosystem

## Let's webize Prolog, prologize the Web, and populate the Web with clever Prolog agents!

(Draft printed 2026-03-22 at 20:50:11)

Springer Nature

# Preface




In this book, we present a proposal for a profile of Prolog called *Web Prolog*. We like to think of it as a *web programming language*, or more specifically, as a web *logic* programming language, but also as a kind of simple web-based *agent* programming language. In addition, the architecture for an extension of the traditional Web that we think of as *the Prolog Web* is described, along with descriptions of some of the *Prolog agents* that might dwell there. Together, Web Prolog, Prolog agents and the Prolog Web form what we shall refer to as the *Prolog Trinity ecosystem* – a niche within the vast ecosystem of the World Wide Web.

As can be gleaned from its title, as well as from the previous paragraph, the Prolog programming language plays the lead role in the book, and Prolog programmers and Prolog system implementers are indeed among its intended readers. However, we do not think a reader has to be a very experienced Prolog programmer or implementer to get something out of it. In fact, we are presenting what we believe is a somewhat novel view of Prolog and Prolog programming, so it might even be good if your mind is a beginner's mind, still open to a somewhat different story.

Another programming language, namely Erlang, has an important role to play in the book, so Erlang developers and other actor-programming aficionados might be interested. Or at least we hope so, as we might have some work cut out for them, both as regards the future specification of the Web Prolog language, as well as for its implementation. Note, however, that although the book is written with an audience of Erlangers in mind, some basic knowledge of Prolog is still assumed.[1]

Logic programming researchers and system developers aiming for logical purity might be interested too, but should be aware that the book is focusing on a non-

[1] For readers lacking the necessary background in Prolog, we provide a few recommended learning resources in Appendix E.2.

declarative concurrency and distribution model for Prolog, and does not purport to contribute to its logic programming aspects. It is still only "good ol' Prolog." Logic programming researchers might find *use* for it, however, as a way to present their own work to the world, and in particular to other Web Prolog programmers. Oh, and we almost forgot, we also intend to show how to wrap the whole globe in pure logic, and logic programming researchers may want to find out what that means.

The book is also written with an audience of Semantic Web researchers and practitioners in mind, in the hope that it will seem relevant to some of them. After all, the Prolog community and the Semantic Web community have something in common, namely *computational logic*, and probably also a deep conviction that, on some level at least, building software agents that are *really* clever requires logic, reasoning and a relational way of thinking. We hope to be able to show that Web Prolog might serve as a *semantic web logic programming language* – a language that fits in very well among the other web logic languages defined and standardized by the W3C, and a suitable language for building semantically aware web applications, such as intelligent web agents.

Readers interested in Artificial Intelligence (AI) might also be able to get something out of the book, at least if they are somewhat familiar with Prolog. In particular, readers who, along with many others, hold the view that a lot of what happens within the field of AI in the coming decades will take place on the Web, and that intelligent conversational agents are likely to serve as the *face* of AI. Note, however, that the book has almost nothing to say about artificial neural networks and other non-symbolic approaches to AI, but is primarily concerned with symbolic approaches, focusing on the use of logic and reasoning for building web agents. Having said that, it is of course impossible to ignore the recent success of so called Large Language Models (LLMs), so the text briefly speculates about the relation between LLM technology and our own approach to symbolic AI on the Web. We can only scratch the surface, but can say already at this point that finding the proper balance between such technologies appears to be as challenging as it is important for the logic programming community going forward.

The categories of potential readers outlined above – Prolog programmers and system implementers, Erlang developers, logic programming researchers, and Semantic Web practitioners – are not merely an audience for this book but also (potentially) the main stakeholders in a broader endeavor: the realization of the Prolog Trinity ecosystem. Each of these groups brings essential expertise and perspectives that contribute to the development, refinement, and practical deployment of Web Prolog, Prolog agents, and the Prolog Web. Together, these stakeholders form a core community that might shape the Trinity, ensuring that it is not only an intellectually compelling idea but also a practical and viable extension of the Web.

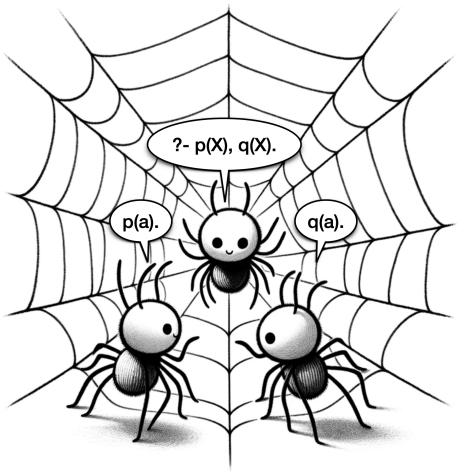
It may all be a pipe dream, but it is not a castle in the air![2]

[2] https://blogs.bmj.com/bmj/2017/01/27/neville-goodmans-metaphor-watch-dream-a-dream-of-ivory-castles-in-the-air/

## Acknowledgements

# Contents

# Chapter 1
# Introduction




This book articulates a *vision* – an ambitious and forward-looking proposal – for a reconceptualization of Prolog in the context of the modern Web. At the heart of this vision lies *Web Prolog*, a carefully crafted profile of the Prolog language, reimagined as a logic-based programming language for the Web. More than just a variant of Prolog adapted for networked environments, Web Prolog aspires to function as a foundational language for a new generation of *web-native logic programming* and *web-based agent systems*.

We propose to think of Web Prolog not only as a tool for building programs that interact over the Web, but as a language for defining logic-driven agents – *Prolog agents* – capable of reasoning, communicating, and coordinating their behavior across distributed contexts. (The term "agent" is used very liberally throughout this book and does not imply agency in its fullest sense.) These agents inhabit a conceptual space we call the *Prolog Web*: an architectural extension of the traditional Web, designed to accommodate structured logical interactions, intelligent agent communication, and declarative knowledge sharing.

Taken together, these three components form the *Prolog Trinity ecosystem*. This ecosystem is envisioned as a niche within the broader World Wide Web, offering a unique paradigm for symbolic AI, declarative communication, and logical coordination on the Web. It represents not merely a technical enhancement, but a philosophical stance: a reassertion of the relevance of symbolic reasoning and declarative knowledge in an era increasingly dominated by opaque statistical models.

This opening chapter serves as a prologue to that larger vision. It prepares the ground for the conceptual foundations, technical proposals, and speculative possi-

bilities explored in the chapters ahead, and it introduces the *what*, *why* and *how* of a project that aims to bring it to life.

## 1.1 The Prolog Trinity ecosystem

> Employed as a language for communicating with computers, logic is higher-level and more human-oriented than other formalisms specifically developed for computers.
>
> *Robert Kowalski*

As suggested by the simple diagram in Figure 1.1, we think of Web Prolog, Prolog agents and the Prolog Web as three different kinds of entities, playing three distinct roles in the comprehensive Prolog-based ecosystem that we think of as the Trinity.

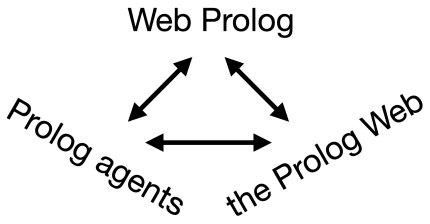



**Fig. 1.1** The Prolog Trinity ecosystem: Web Prolog, Prolog agents, and the Prolog Web, and their mutual dependencies.

The diagram illustrates the main idea on the highest level of abstraction we shall care about. The arrows are meant to indicate the many interdependencies and symbiotic relationships that exist between these entities, and to remind ourselves that we are looking at one and the same ecosystem from three different angles.

Put simply, on this level of abstraction the Prolog Web is a network of Prolog agents being programmable in Web Prolog and talking Web Prolog to each other. Expressed more elaborately, the Prolog Web is populated, and indeed also propped up, by Prolog agents, and provides, in the form of distributed Web Prolog databases, some of the knowledge that these agents need to do their thing. Prolog agents in the Prolog Web environment interact with other Prolog agents, using Web Prolog as their language of communication. They are also able to talk with (other kinds of) agents outside this environment such as humans, maybe through a conventional graphical user interface (a GUI), or perhaps using spoken natural language in a voice user interface (a VUI), and maybe even in a dialog with an *embodied conversational agent* (an ECA).

It is important to understand that the Prolog Trinity ecosystem is not an island. The Prolog Web does not exist in complete isolation, cut off from the rest of the Web. Rather, it is meant to serve as an *extension* of the traditional Web, along with other extensions such as the Semantic Web and the Web of Things. Just like these, the Prolog Trinity can be regarded as a *sub-ecosystem* of the Web ecosystem in its entirety. As we shall see, it can also be thought of a sub-ecosystem of the

larger ecosystem surrounding Prolog – the already existing Prolog systems and the communities of logic programming researchers, system implementers, and other users of Prolog.

## 1.2 Walking in the footsteps of inventors

Before we continue, let us first pay tribute to five people who invented technologies that form the basis for the work presented in the book and laid the groundwork for the Prolog Trinity ecosystem. We have put them side by side in Figure 1.2.

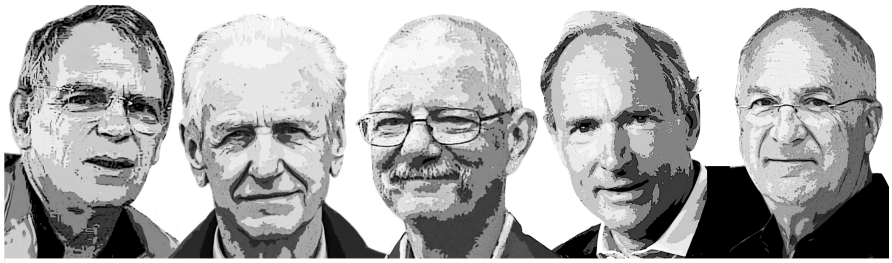


**Fig. 1.2** Five inventors whose ideas underpin the Trinity: Colmerauer, Kowalski, Armstrong, Berners-Lee, and Harel (photo montage).

On the left we find Alain Colmerauer. He invented Prolog in 1972 and built the first implementation. Sadly, he passed away in 2017. To the right of him is Robert Kowalski – who came up with a lot of the theory behind Prolog which inspired Colmerauer. He is still around, and is still doing a lot of logic programming research and development. Third from the left, we find Joe Armstrong, who invented another programming language called Erlang. He liked Prolog, and to some extent, as we shall see, Erlang was inspired by Prolog. Regretfully, Joe Armstrong died in 2019. To the right of him, we find Tim Berners-Lee. He invented the Web, the first web browser, and the protocols and algorithms allowing the Web to scale. He is still very much alive and active trying to build a better Web. Finally, on the right, we find David Harel, the inventor of Statecharts, a rather succesful visual state-machine programming language. Why his work has a role to play in the Prolog Trinity ecosystem will be revealed later in the book.

These five inventors are (or were, for those that are no longer with us) scientists and engineers who have made significant contributions to the field of computer science. We do not mean to suggest that they were alone in inventing these technologies.[1] The work presented in this book is indebted also to the many, many scholars who evolved the original ideas into the shape they have today. This book tries to build on their achievements, combining chunks of their work into a what we hope will emerge as a coherent whole, and preferably a whole greater than the simple sum of its parts.

[1] For example, Robert Virding and Mike Williams were heavily involved in the design and development of Erlang.

## 1.3 The Prolog Trinity ecosystem concretized

A deeper understanding of what the Prolog Trinity is all about can hopefully be gained by considering the tiny partition of the Prolog Web suggested in Figure 1.3.

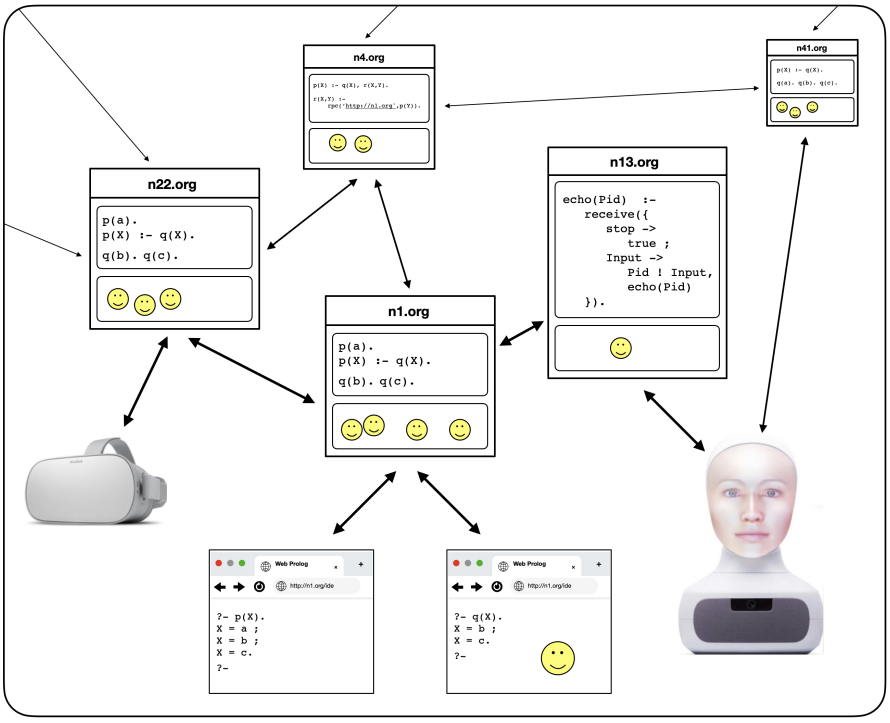



**Fig. 1.3** End-user clients interacting with Prolog agents within a tiny partition of the Prolog Web.

On this fairly low level of abstraction, the Prolog Trinity ecosystem can be understood as a concrete network of concrete *Prolog nodes*. In Figure 1.3, nodes are shown as boxes labeled by a *domain name* at the top. The arrows symbolize connections between different nodes, and between end-user clients and nodes. As suggested by the diagram, each node is divided into two "containers," one container dedicated to the management of actor processes, the other to the storage of Web Prolog programs and data.

### 1.3.1 Prolog nodes and Prolog actors

Inspired by Erlang's notion of an actor, the *Prolog actor* is the fundamental unit of computation – and the simplest type of agent – in the Prolog Trinity ecosystem. The smiley emojis in Figure 1.3 represent Prolog actors of various kinds that live and work at nodes. When a node is computing something for a client, it is often one or more of these actors that are responsible, even if this is not evident from the outside. Hiding an actor behind an emoji makes it look simpler than it actually is, and in fact, an actor has an interesting internal structure. The diagram in Figure 1.4 shows three actors, one of which has just been spawned and one that has been opened up so that we can look inside (where, at this level of abstraction, they all look the same).

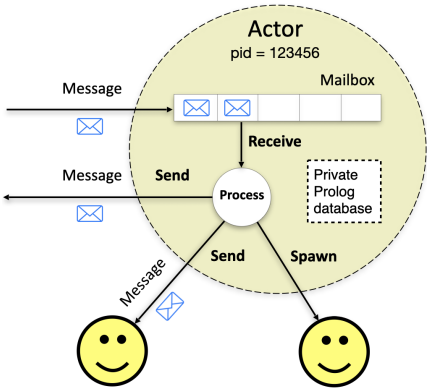



**Fig. 1.4** The anatomy of a Prolog actor.

A Prolog actor is a *process*, capable of communicating with the world around it through messaging. An actor has a unique address – a *process identifier*, or *pid* for short. If we have the pid of an actor, we can message it, and since pids can themselves be part of a message, it allows other actor processes to communicate back. Furthermore, an actor has a *mailbox* that stores incoming messages. There is a *receive* operation that allows the actor to select and process messages from the mailbox, and a *send* operation that can be used to send messages to any other actor. There is a *spawn* operation that allows an actor to create other actors – child actors as it were. Finally, every actor is equipped with a private Prolog clause database, a dedicated space where programs and data exclusive to that actor are stored.

Authorized end-user clients can connect and interact with nodes. These clients may range from web browsers running simple JavaScript-based Prolog terminals or intelligent web applications, to more sophisticated devices like VR headsets or social robots. (Note that not all clients are end-user clients because nodes can be clients too.) Human users interacting with end-user clients typically trigger activity in the Prolog Web, where agents as well as connections between agents are more or less

frequently being created, interacted with, and then destroyed. Some agents and some connections live for a long time, while others may only last for fractions of a second. This is obviously a form of dynamics that is not easy to capture in a static diagram such as Figure 1.3.

A node can be seen as a special-purpose web server serving data in the form of Prolog terms (possibly encoded in JSON) to clients as well as to other nodes. Just like a web server, a node has a domain name, makes use of HTTP and/or WebSockets, and is usually capable of serving more than one client at the same time.

We use the term *node* rather than *server*, partly because this is what Erlang uses, but a more important reason is that a node is much *more* than just a web server. A node is also a *runtime system* for Web Prolog capable of executing Web Prolog programs stored by the node and/or passed to it by the client over the web APIs. The Prolog Web is a network of such code, ready to be executed by actors, ready to be queried by clients.

### 1.3.2 Web Prolog programs and data

Data and programs stored at a node come in two flavors, characterized by two kinds of Web Prolog clauses. One flavor might have clauses looking like this:

```
mortal(Who) :- human(Who).
```

Clauses of this flavor have logical interpretations, and this particular clause translates into the formula $\forall x[human(x) \rightarrow mortal(x)]$ in first-order predicate logic.

As suggested in Figure 1.3, a node can be queried from a terminal running in a web browser. Given our definition of `mortal/1` and a set of Prolog *facts* such as

```
human(socrates).
human(plato).
human(aristotle).
```

we can ask for the solutions to a query `?-mortal(Who)`. When we run the query from a terminal we enter into a kind of interaction with a Prolog process that most readers are probably familiar with – in which the system only offers one solution at a time and the user may request further solutions by typing a semicolon:

```
?- mortal(Who).
Who = socrates ;
Who = plato ;
Who = aristotle.
?-
```

Such queries can also be given logical interpretations, and this particular one translates into the formula $\exists x[mortal(x)]$. The interaction not only shows us that there exists an $x$ such as $x$ is mortal, but also provides us with examples. The point here is

to remind ourselves that Prolog is a *logic programming language* eminently suitable for *knowledge representation*, or for other uses that require logic. The interaction also demonstrates that Prolog is a *nondeterministic* programming language, meaning that a query may have several solutions, which can be lazily iterated over using Prolog's backtracking mechanism.

In comparison with the clause defining the predicate `mortal/1`, the following clause has a very different flavor and looks a lot more like a program, and indeed something from the Erlang world – a clause in the *style* of Erlang:

```
echo_actor :-
    receive({
        echo(From, Msg) ->
            From ! echo(Msg),
            echo_actor
    }).
```

As its name suggests, it implements the core of an echo server: an actor capable of receiving messages and returning them to the sender as an "echo." In contrast to the definition of `mortal/1`, this clause cannot be given a logical interpretation; both `receive/1` and `!/2` are built-in predicates with side effects, so the clause must be understood as a procedural construct rather than a declarative one – a procedure that, when executed by an actor process, determines that actor's *behavior*.

The predicate `spawn/2` creates a concurrent process running in parallel with the caller, binding the variable in its second argument to a process identifier:

```
?- spawn(echo_actor, Pid).
Pid = 10783471.
?-
```

An interaction exercising the echo actor might look like this:

```
?- self(Self).
Self = 81902375.
?- 10783471 ! echo(81902375, hello).
true.
?- receive({Message -> true}).
Message = echo(hello).
?-
```

Readers familiar with Erlang will recognise the pattern. At this point other readers only needs to know that the predicate `self/1` retrieves the pid of the calling process, `!/2` sends a message to another process, and `receive/1` waits for an incoming message. Even this tiny example already conveys the essential properties of an actor. It has its own mailbox, reacts asynchronously to messages, can run indefinitely, and maintains identity through its process identifier.

The spawn operation can be configured by means of *options*, so there is a `spawn/3` as well, where the optional last argument is a list of options. For example, the `node`

option takes a URI and allows an actor to be spawned on the remote node to which that URI points:

```
?- spawn(echo_server, Pid, [
       node('http://n4.org')
   ]).
Pid = 41983256@'http://n4.org'.
?-
```

This demonstrates that our actor model is not just for local concurrency but is inherently network-transparent and distributed – this is the "Web" in "Web Prolog."

Other options inject source code into the spawned actor's private database, or specify what should happen when the actor dies, i.e. what its parent (if it has one) might learn about the reason for its death, and how its children (if it has any) will be treated when it terminates. These aspects of actor lifecycle management will be explored in greater depth later in the book.

Although the echo actor is intentionally simple – a specialized process whose only task is to reflect incoming messages back to the sender – it represents only the most basic form of actor behaviour. Through the combined use of options and meta-programming techniques, Web Prolog, like Erlang, supports a wide range of *generic* behavioural patterns. These include actors that implement a generic client-server model, actors capable of actors supervising other actors, and actors that function as event-driven, hierarchical state machines. We shall return to these behavioural patterns when we discuss server templates, supervision trees, and statechart actors in later chapters – including several powerful patterns that cannot easily be expressed in Erlang but arise naturally from properties that Prolog, but not Erlang, has.

## 1.4 The current state of Prolog

Created in 1972, Prolog is a rather old programming language. Any rumors of its imminent death can easily be debunked, however, as it is currently experiencing something of a revival. New systems are being implemented; features such as tabling and constraint-logic programming libraries are being added to both new and established systems; new books are being written; the community is small but thriving, and the language is still taught at some universities.

Still, the current state of Prolog is far from ideal. The community is *too* small and lacks the cultural visibility that defines a healthy, growing language ecosystem. This is well known and was fairly recently documented in a series of papers.

In 2022, as part of the celebration of the Year of Prolog,[2] a number of logic-programming scholars published a survey paper titled *Fifty Years of Prolog and Beyond* (hereafter referred to as FYPB), as well as a book of papers titled *Prolog: The Next 50 Years* (henceforth PN50Y), which followed a year later. As they are

[2] `https://prologyear.logicprogramming.org`

written by a substantial portion of the *crème de la crème* of the Prolog community, it makes sense to take these papers as probably the best and most current descriptions of the history of Prolog, its current state, and what the authors wish for its future. While this book does not discuss the history of Prolog, its *current* state and its future are certainly relevant.

What makes FYPB and PN50Y suitable as background for the project described in this book – and the reason we shall refer to them at various points going forward – is that we share a common interest with their authors: to improve Prolog and its ecosystem as a whole. We ask ourselves the same questions they pose in the following passage from FYPB:

> How might Prolog and its community evolve in the future? Can we better unify the new aspects that are offered by different implementations? How should efforts for increased portability be organized? Does it make sense to aim for a unified language? And what tools could be provided to ease development? Furthermore, we propose a plan for future steps that need to be taken to evolve Prolog as a language.

However, as will become apparent, we arrive at different answers and at different ideas for how things might be improved. The Prolog Trinity ecosystem should be seen as our overarching proposal for these various ideas – a scaffolding that helps ensure that they are mutually coherent and consistent, improves the cultural visibility of the current Prolog ecosystem, steadies its state, and helps heal what is broken.

## 1.5 Motivating the Prolog Trinity ecosystem

A vision may be coherent, a design elegant, and an implementation technically feasible, yet this alone does not suffice to justify its realization. What is also required is a clear motivation and a compelling argument for why the undertaking would be worthwhile.

The purpose of this book is to provide such an argument, and to show that its realization would not only benefit the implementers and users of existing Prolog systems, but also strengthen the Prolog ecosystem and the community as a whole. The following subsections outline the main motivations for developing the Prolog Trinity ecosystem.

### 1.5.1 Why Web Prolog?

1. **A simple model for concurrency-oriented programming.** Web Prolog brings a clear and comprehensible model of concurrent programming to the Prolog world. Concurrency has long been a source of complexity within the Prolog community, and the ISO threads model offers only a fairly low-level solution. By contrast, extending Prolog with Erlang-style constructs yields an elegant and expressive multi-paradigm language – one that can, in many respects, be more

capable than traditional Prolog or Erlang alone. The actor model provides a high-level alternative for structuring concurrent programs, with semantics that are simple enough to teach and reason about. Another practical advantage is that Erlang-style concurrent programming in Web Prolog can be learned from the substantial literature produced by the Erlang community.

2. **An elegant model for distributed programming.** Erlang introduced a remarkably clear and effective model for distributed programming, based on the principle that communication between processes should be independent of their physical location. Web Prolog adopts this foundation while embedding it in a logic programming context. It treats distribution as a natural extension of local message passing: actors communicate across nodes with the same simplicity and uniform semantics as within a single node. By preserving Erlang's conceptual clarity and pragmatic approach to fault handling, Web Prolog offers a transparent model of distributed computation – one that combines the expressive reasoning of Prolog with the robust, process-oriented discipline of Erlang.
3. **A profile of Prolog for web programming.** Although Prolog extended with Erlang-style concurrency primitives already forms a capable general-purpose language, Web Prolog focuses on a more specific goal: to define an accessible and coherent *profile* of Prolog for programming the Web. Two complementary modes of interaction are supported: declarative query answering over stateless or semi-stateful HTTP, and stateful actor communication over persistent WebSocket connections. Drawing on Erlang's distributed model yet reimagining it for an open, heterogeneous, and loosely federated environment, Web Prolog provides a foundation for scripting the Web with logic and for implementing communication protocols among distributed agents. In other words, Web Prolog is intentionally bilingual at the transport level while remaining monolingual at the language level: the same Prolog terms and the same programming idioms scale from request–response querying to long-lived actor conversations. Together these mechanisms cover both short-lived and enduring forms of interaction, making Web Prolog a versatile and elegant language for the programmable Web.

### 1.5.2 Why Prolog agents?

1. **Agents as powerful programming abstractions.** Prolog agents provide a simple yet expressive abstraction for continuous and interactive behaviour. A Prolog agent is a software process capable of communicating with other agents on the Prolog Web, exchanging Prolog terms of arbitrary complexity and, when appropriate, fragments of Web Prolog programs themselves. In this book we treat both actors and nodes as agents, and both come in specialised varieties such as toplevel actors, shells, statechart actors, and nodes with different profiles and capabilities. Because agents can be queried, instructed, supervised, and composed into larger systems, Web Prolog becomes not only a language for logic

programming but also a language for *agent programming*, supporting the development of single agents as well as broader multi-agent systems. The result is a unified abstraction for autonomous, interactive components at the centre of the ecosystem.

2. **Statechart actors.** Statecharts, introduced by David Harel, offer a visual formalism for describing complex, reactive state machines with hierarchy, deterministic parallelism, and well-defined event semantics. Statechart actors are Prolog agents that provide a principled and highly expressive model for event-driven control. When viewed as actors, statechart processes become natural participants in a message-passing environment. Using Web Prolog as its data modelling and scripting language allows statechart actors to integrate declarative reasoning with robust, event-driven control. This combination yields a well-suited platform for constructing AI applications of many kinds, particularly intelligent conversational or embodied agents. Incorporating statechart actors broadens the ecosystem's ability to support sophisticated reactive systems. The point is not to add another agent type, but to make *control* a first-class, inspectable artefact in the same ecosystem as declarative models and message-passing interaction.
3. **Turn-key Prolog nodes.** An important practical motivation for the Prolog agent model is the availability of *turn-key Prolog nodes* – ready-made, nearly complete applications that become operational as soon as domain logic and configuration are supplied. Turn-key nodes lower the threshold for participation by capturing proven patterns for supervision, communication, persistence, and deployment, thereby sparing developers from boilerplate concerns and allowing them to focus on domain-specific reasoning and interaction. Because these nodes act as agents in their own right, they can be queried, extended, and orchestrated alongside user-defined agents, making the environment uniform and easy to understand. Such simplicity and accessibility are essential if the Prolog Web is to grow into a living shared environment rather than remain a conceptual construct.

### 1.5.3 Why the Prolog Web?

1. **A network of logic-driven Prolog nodes.** A central motivation for the Prolog Web is that a Prolog node is not a monolithic service but a participant in a distributed network of message-passing, logic-driven agents. Turn-key Prolog nodes provide the structure needed to populate this network: self-contained, declarative services that embody the communication, reasoning, and supervision patterns of Web Prolog. Nodes share a uniform design so that newcomers can learn them quickly, practitioners can use them effectively, and deployers can operate them with minimal effort. This consistency allows the Prolog Web to grow organically into a coherent, agent-populated environment rather than a collection of isolated services.
2. **A natural foundation for modern AI.** Viewed in this light, the Prolog Web is not merely an attractive programming environment but a natural foundation

for building modern AI systems. Logic-based, message-passing agents are well suited to the construction of hybrid and classical multi-agent systems, distributed reasoning frameworks, and embodied conversational agents that require explainability, coordination, and long-lived interaction. The Prolog Trinity ecosystem aligns these requirements with web-native infrastructure, providing a principled model for constructing AI that is transparent, distributed, and inherently capable of reasoning. This makes the Prolog Web an appealing platform for developing the emerging *Agentic Web* (a buzzword for a web of autonomous, reasoning agents).

3. **Nondeterministic RPC as a web-programming capability.** Another motivation for the Prolog Web is that it supports forms of distributed interaction that are difficult to realise cleanly in other web programming models. In particular, its notion of nondeterministic remote procedure calls allows a client to request multiple answers to a query from a remote node, mirroring Prolog's local search behaviour across the network. This preserves the declarative reading of logic programs while extending it seamlessly to distributed settings, enabling agents to cooperate in solving search problems, generating alternative plans, or exploring hypotheses in parallel. Treating nondeterministic RPC as a first-class web capability makes the Prolog Web a distinctive and expressive environment for distributed logic programming.
4. **Federation with explicit boundaries.** The Prolog Web is designed as a loosely coupled federation: nodes are free to specialise internally, but must expose small, well-specified interfaces at the boundary. This matches how the Web scales, and it also creates a natural place for capability restrictions, resource limits, and ownership policies – concerns that become unavoidable once we take openness seriously.

### 1.5.4 Why the Prolog Trinity ecosystem as a whole?

Figure 1.3 does not reveal anything about the *implementation* of the nodes it depicts. They could all be implemented on top of a single Prolog system, or several systems may be involved, perhaps one for each node, reflecting the diversity of the Prolog landscape. Building the Prolog Web on top of one system is probably the fastest path to a public demo, early feedback, and something that runs on today's Web and demonstrates value. In a research-oriented community such as Prolog's, stopping at that stage would not be unusual. Our proposal, however, is to turn the endeavour into a collective, community-driven project.

The Prolog Trinity ecosystem is therefore motivated by the need for a common scaffolding that supports, rather than suppresses, the existing diversity of Prolog systems. Different implementations excel in different domains: some prioritise performance, others constraint solving, tabling, or integration with foreign languages. Rather than seeking unification through a large monolithic standard or a single system, we treat this diversity as a resource. Web Prolog is defined as a small,

implementable, web-aligned fragment that acts as a *lingua franca* for cross-system interoperability. By wrapping each system in a node that speaks Web Prolog, heterogeneous engines can collaborate in real time without surrendering their strengths. This is not a call for unification by portability alone; it is a call for precise, web-aligned interfaces among nodes using small, well-specified protocols. Modest systems can implement just the essential subset and still participate fully. The result is a federation that mirrors how the Web itself grows: loosely coupled, strictly specified at the boundaries, and free to specialise behind them.

To move from the narrow view of a single implementation to the broader perspective of the Prolog ecosystem, we must also relate our aims to concerns already articulated by the community. As we learned above, FYPB and PN50Y ask how Prolog might evolve, how portability might be improved, whether a unified language is sensible, and what tools would best support development. These questions form a shared point of departure, yet our proposed answers differ in important ways. Where FYPB sees weakness in "limited portability", we concentrate on a portable, carefully defined *fragment* of Prolog as a basis for interoperability. Where they see a problem of *fragmentation*, we see an opportunity for *diversification*, tied together by a clear Web Prolog substrate. Where they identify a threat in the "post-desktop world of JavaScript web applications", we see an opportunity to design a logic-based, actor-oriented, federated programming environment capable of thriving on the Web itself.

Our programme complements the forward-looking agendas in FYPB and PN50Y: where they outline desirable directions for the future of Prolog, we propose a web-native and agent-centric realisation path, complete with concrete artefacts that can be implemented, tested, compared, and incrementally standardised. For these reasons, we argue that the realisation of the Prolog Trinity ecosystem is not merely feasible but worthwhile: it offers simplicity for developers, coherence for the community, and a principled platform for the next generation of AI on the Web.

Finally, a note on scope. The Trinity is not proposed as a replacement for existing Prolog systems, but as a way to make them cooperate. It is not a demand for a monolithic unified Prolog, but an attempt to define small boundaries that many systems can implement. And it is not a promise of a frictionless open network; rather, it is an argument that openness becomes manageable once capabilities and interfaces are explicit.

## 1.6 Webizing Prolog and prologizing the Web

> The essential process in webizing is to take a system which is designed as a closed world, and then ask what happens when it is considered as part of an open world. Practically, this effect on a computer language is to replace the names/tokens/identifiers for URIs. Thus, where before reference could only be made to something in the same document/program/module one can with equal ease make reference to something in a different one somewhere in that abstract space which is the Web.
>
> *Tim Berners-Lee, 1998*

Some thirty years in the making, the World Wide Web is a formidable success and the biggest distributed programming system ever constructed, and has, over the years, become a key delivery platform for a variety of sophisticated interactive applications in just about any conceivable domain.

The traditional web, already in its most primitive form as a Web of Documents (also known as Web 1.0), is *distributed* (running on more than one machine), *decentralized* (running at different geographical locations and/or having different owners) and *open* (anyone can use and contribute). These are some of the traits the design of Prolog Web seeks to inherit from the conventional web. Whereas distribution comes with the territory, decentralization and openness have more to do with *web culture*.

### 1.6.1 Webizing Prolog

To develop the design of the Prolog Trinity ecosystem we need to apply the process of *webizing* to Prolog. Here is what we need to do, and each of the points must be meticulously addressed:

1. **Introduce URIs into the ecosystem**: Uniform Resource Identifiers (URIs) are fundamental to the web as they uniquely identify resources. Integrating URIs into Prolog will enable the language to directly address and interact with the plethora of resources available on the web.
2. **Exploit the existing web infrastructure**: The Web's infrastructure includes protocols such as HTTP and WebSocket as well as data formats such as JSON. Web Prolog should be able to utilize these protocols and data formats seamlessly, and as much as possible behind the scenes.
3. **Make use of existing means for security**: Web security is a multifaceted domain, encompassing aspects like authentication, authorization, data integrity, and confidentiality. Web Prolog needs to integrate with existing security mechanisms such as TLS for encrypted communications, OAuth for authorization, and possibly more sophisticated web-specific security protocols. These mechanisms must be ingrained within the Web Prolog's runtime environment to protect against common web vulnerabilities.
4. **Ensure web-size scalability**: The scalability challenge for Web Prolog is twofold: it must handle large numbers of simultaneous connections and manage extensive datasets. This will require efficient multitasking and concurrency models, optimized algorithms for distributed computing, and an architecture that allows for load balancing and a lot of caching.
5. **Support web application development**: To truly support web application development, Web Prolog should offer a development experience that is both powerful for experienced developers and accessible to those new to Prolog.
6. **Ensure fitness to web culture**: The web has a distinct culture that prioritizes openness, collaboration, and community-driven development. Web Prolog should embrace this by being open source, having a clear and liberal licensing

model, providing excellent documentation and support, and fostering a healthy community around it.
7. **Aim for standardization**: Finally, to gain widespread adoption, Web Prolog must strive for standardization. This entails working with standard bodies and the broader community to establish an official specification for Web Prolog. Standardization helps ensure interoperability between different implementations and provides a stable foundation upon which developers can build.
8. **Play nicely with existing web standards**: For Prolog to be web-friendly, it must enter into fruitful relations with existing W3C standards. This implies that Web Prolog must be able to handle standard web protocols and document formats, work efficiently with web APIs, and follow guidelines such as for internationalization. Prolog's strength in dealing with ontologies and rule-based systems allows it to play nicely with the existing standards for the Semantic Web. We believe we may also be able to connect to a W3C's SCXML, although this may be harder. More on this later.

### 1.6.2 Prologizing the Web

Webizing Prolog involves adapting and extending Prolog to operate effectively in the context of the Web. The focus is on Prolog and the task of designing and developing a special-purpose profile that makes it more useful for web programming than it currently is. The phrase "prologizing the Web" is just another way to describe the very same process, aiming at bringing Prolog and the Web closer together. Prologizing the Web involves extending the traditional web with the capabilities of Prolog. We put the Web in focus and seek to improve and further evolve it by the addition of Prolog. We do this because we believe that a prologized Web is better than a Web not prologized.

1. **Ensure clever use of Prolog's capabilities**: Make sure that the capabilities that Prolog has, such as knowledge representation, reasoning, problem solving and parsing can be put to good use on the Prolog Web.
2. **Leverage the power of Erlang-style concurrency:** Ensure that the actor-based predicates for process creation and message passing can be used to support concurrency and distribution across the Web.
3. **Ensure that Web Prolog runs in web browsers**: Do not introduce constructs that make it impossible to implement Web Prolog in browsers.

Towards the end of the book, in Chapter **??**, we shall return to these points, and assess what has been done to webize Prolog, prologize the Web and create a design for the infrastructure supporting the Prolog Trinity ecosystem.

## 1.7 Aiming for a standard

The exclamation mark at the end of the book's subtitle might lead some readers to suspect that it can be read as something of a manifesto or mission statement. There is indeed some truth in that, as we are trying to draw the intellectual support necessary for an attempt to create *web standards* for the technologies relevant to the realization of the Prolog Trinity ecosystem. Presumably, such an attempt can only succeed if supported by a reasonably large number of individuals from the relevant communities – by people from the Prolog and Erlang communities, people in the Semantic Web community, and people interested in AI and in building conversational systems.

## 1.8 Thirty years of Prolog on the Web

We do not want to give the impression that the suitability of Prolog for programming the Web has not been investigated before. On the contrary, in the mid-1990s the advent of the Web sparked a great deal of interest in the logic programming community, reflected in two workshops devoted to the subject and in the development of libraries that mostly relied on creating so-called CGI programs in Prolog. Here is a snippet of text lifted from the call for papers for the second workshop:[3]

> This workshop is the second in a series intended to explore the elective affinities between Logic Programming and Internet technologies with emphasis on enhancing the World Wide Web with knowledge, deductive abilities and superior forms of interactive behavior. With the paradigm shift to highly inter-connected computers and programming tools, logic programming languages have a unique opportunity to contribute to practical Internet application development. Simplicity, remote executability, robustness, automatic memory management, are among the features some LP languages share with emerging tools like Java. Superior meta-programming and high-level distributed programming facilities, built-in grammars and dynamic databases, declarative semantics are among their competitive advantages.

Since then the Web has developed at an amazingly fast pace, and web applications have become increasingly speedier and more interactive. The Web of today is *very* different from what it was almost thirty years ago – faster, more reliable, more mobile, and with a fantastic selection of browser APIs that lets a web developer add features such as spoken interaction, video or 3D graphics to an application. There is a reason we nowadays talk about, not only the Semantic Web, but also the Conversational Web, the Mobile Web, the Web of Things, and (more recently) the Agentic Web.[4] Furthermore, many (potentially) disruptive technologies such as automation, artificial intelligence, blockchain technology, wearable technology, augmented reality, virtual reality, 5G, 3D printing, robotics and quantum computing are likely to more or less depend on and be integrated into the Web. In this book we focus on artificial intelligence and its integration into the Web.

---

[3] `http://www.cliplab.org/lpnet/proceedings97/preface.html`

[4] See (Lager and Myrendal, 2012) for a description of the many facets of the current Web.

With the Web (technically) so much better and so different from what it used to be, there is, in our opinion, every reason to make a new attempt at "enhancing the World Wide Web with knowledge, deductive abilities and superior forms of interactive behavior." In particular, "superior forms of interactive behavior" is likely to depend on the WebSocket protocol and/or the implementation of Web Prolog in browsers or (mobile) apps. The WebSocket standard did not exist twenty years ago, which meant that every approach to web logic programming had to work solely with HTTP. While HTTP remains the most important protocol (and as we shall see, two out of the three APIs driving the Prolog Trinity ecosystem rely on it), important aspects of some of the inventions described in this book rely on the WebSocket protocol, a so called server-push technology which opens a whole slew of opportunities for developers of web applications based on Prolog.

Regarding the history of Prolog, we do not suggest that nothing has happened between now and then. From the 1990s onward, Prolog has repeatedly adapted to the web's changing landscape. Early work such as the CLIP group's PiLLoW library showed how HTML and HTTP could be handled declaratively by treating web structures as Prolog terms. With the rise of the Semantic Web in the 2000s, Prolog gained renewed relevance: SWI-Prolog's RDF store, web libraries, and the ClioPatria platform demonstrated that a Prolog system could serve simultaneously as a high-performance knowledge base and a full HTTP server.

In the 2010s, more interactive and distributed styles of web programming led to further innovations. Pengines (Lager & Wielemaker) made server-side Prolog engines accessible from JavaScript through a simple RPC model, enabling incremental solution streaming and smooth integration with modern web applications. At the same time, projects such as Tau Prolog brought ISO-style Prolog directly into the browser, allowing entirely client-side logic applications written in JavaScript.

More recently, WebAssembly has made it possible to run full Prolog systems inside the browser. SWI-Prolog can now be compiled to WASM via Emscripten, and lightweight systems such as Scryer and Trealla have been ported as well. These provide fast, sandboxed Prolog engines that interoperate smoothly with JavaScript, opening the way for high-performance logic programming in modern web applications.

Among Prolog systems, SWI-Prolog in particular has all the right tools for building server-side support for web applications, in the form of mature libraries for protocols such as HTTP and WebSocket, and excellent support for formats such as HTML, XML and JSON. Therefore, it might be argued that since Prolog can and has been used for building server-side support for web applications, it should already be counted as a web programming language. But since this is true also for Python, or Ruby, or Java, or just about any other general-purpose programming language in existence, it would make the notion of a web programming language more or less empty.

We could of course argue that SWI-Prolog is much better at providing what is needed, but we shall go further than that. We believe that Prolog should claim an even more prominent position in this space. Web Prolog should be used not only for building server-side support for web applications, but should be considered as *part*

of the Web, in much the same way as HTML, CSS, JavaScript, RDF and OWL are parts of the Web. In other words, as we think it deserves it, we would like it to be recognized as *a web technology in its own right*.

## 1.9 The structure of the book, and a pointer to its home page

In the hope that it might help the reader navigate the book's most important ideas and their connections, the mindmap in Figure 1.5 expands the exposition of the three most important aspects of the Prolog Trinity ecosystem.

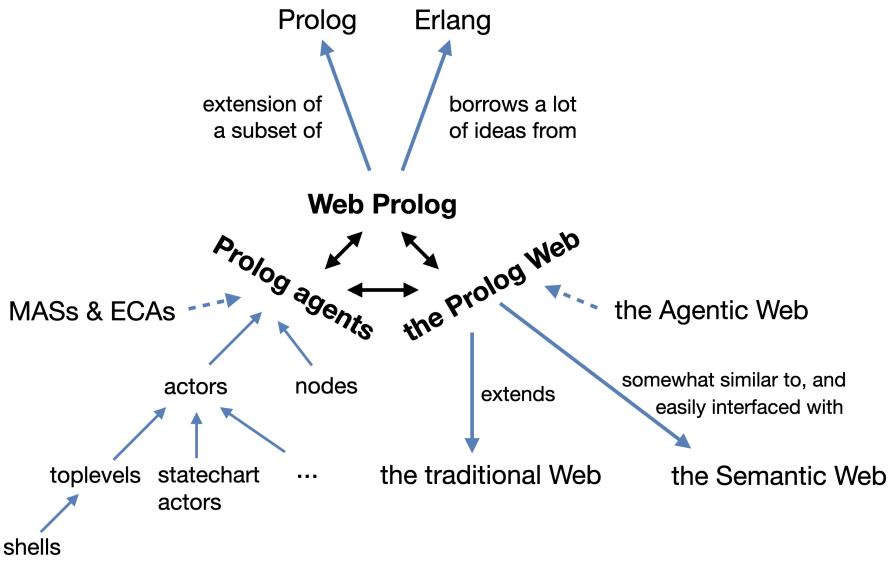



**Fig. 1.5** Mind map of the Prolog Trinity ecosystem and its main conceptual threads.

### 1.9.1 The structure of the book

The mindmap reflects the structure of the rest of this book as follows.

## Part I: The conceptual and architectural core

**Chapter 2** introduces Web Prolog as a Prolog-based, multi-paradigm language for concurrent and distributed programming. It explains what is inherited from standard Prolog, what is added to support actor-style concurrency, and why Erlang is a particularly useful reference point. To make the comparison concrete, the chapter translates a range of Erlang programs into Web Prolog and uses these translations to clarify the intended operational behaviour.

**Chapter 3** develops the agent strand of the mindmap. It introduces a simple taxonomy in which Prolog actors and Prolog nodes are the primary categories, and it focuses on the two agent forms that matter most for the rest of the book: the Prolog toplevel actor and the Prolog node. The chapter describes the built-in interaction protocol that distinguishes a toplevel actor from other actors, breaks a node down into its main components (controller, Web APIs, and the node-resident Prolog database), and illustrates the intended client–node interaction patterns through a series of scenarios. Statechart actors are treated separately in Chapter 6.

**Chapter 4** presents the architectural proposal for the Prolog Web and shows how multiple nodes can be composed into an open, decentralised network. A central theme is distributed programming with an emphasis on programming models that aim to be close to network-transparent when appropriate. The chapter introduces `rpc/2-3` as a user-facing construct for nondeterministic remote procedure calls and uses it to delineate a sequential partition of the Prolog Web alongside the concurrent, mailbox-driven partition. It also makes explicit several axes along which designs can vary, including concurrency vs. sequentiality and purer vs. less pure interaction styles.

## Part II: Behaviors and other generics

**Chapter 5** implements a set of generic constructs that capture reusable actor patterns. These generics can be specialised or configured after creation-time and encapsulate recurring concerns such as message protocols, supervision, and fault handling.

**Chapter 6** introduces an additional category of actor, termed the *statechart actor*, which represents an advanced form of event-driven state machine. The distinctive feature of a statechart actor, as compared to a conventional actor, is its implementation in a variant of an XML-based statechart language. In this paradigm, Web Prolog is employed solely for data modeling and scripting purposes, rather than serving as the primary programming language for the statechart logic.

### Part III: The broader technological contexts

**Chapter 7** presents a detailed comparative analysis between the Prolog Web and the Semantic Web. In this chapter, we contend that the Prolog Web can be effectively constructed on top of the Semantic Web. Both frameworks are fundamentally grounded in computational logic; however, the Prolog Web, viewed as a relational logic-based environment, exhibits a natural affinity for integration with the Semantic Web. This intrinsic compatibility positions the Prolog Web as a promising candidate for Semantic Web programming, thereby enabling the development of applications that leverage the strengths of both paradigms.

**Chapter 8** turns to AI applications, with particular attention to conversational and potentially embodied agents operating on the Web. It uses this setting to explore how statechart-style control structure and Web Prolog-style symbolic scripting can be combined in a pragmatic architecture for building interactive, tool-using agents.

### Part IV: What's in it for you and your community?

**Chapter 10** asks what the Trinity offers to individual users and to the Prolog community as a whole. It frames the contribution in terms of practical capabilities, interoperability, and ecosystem-level opportunities, and it discusses how some commonly identified risks for Prolog might be mitigated.

**Chapter 11** asks the corresponding question for neighbouring communities, including those approaching the problem from distributed systems, Web engineering, and AI perspectives.

### Part V: Final words

**Chapter 12** moves from vision to execution by outlining an implementation path, identifying dependencies and engineering constraints, and mapping the proposal onto a concrete roadmap.

## 1.9.2 The book's home page

The material presented here is accompanied by a home page for the Prolog Trinity ecosystem, where updated code, examples, draft specifications, and links to related projects will be made available. A link to the home page is here (but not yet...):

`http://trinity.elfenbenstornet.se`

# Part I
# The conceptual and architectural core

# Chapter 2
# Web Prolog

> Imagine a dialect of Prolog with actors and mailboxes and send and receive – all the means necessary for powerful concurrent and distributed programming. Alternatively, think of it as a dialect of Erlang with logic variables, backtracking search and a built-in database of facts and rules – the means for logic programming, knowledge representation and reasoning. Also, think of it as a web programming language, and as a *lingua franca* for logic-based programming systems, standardised by the W3C. This is what **Web Prolog** is all about.
>
> *Web Prolog – the elevator pitch*

## 2.1 The essence of Prolog

Based on formal logic, a subject dating all the way back to the antiquity and tried and tested by generations of logicians and philosophers, logic programming forms a paradigm of it own, very different from the imperative or functional programming paradigms. Prolog is generally regarded as the first logic programming language, and is arguably the most important one. Consider the following program:

```
husband(Wife, Husband) :- wife(Husband, Wife).

wife(socrates, xantippa).
wife(aristotle, pythias).
```

We have here a *rule* that translates into the following formula in predicate logic:

$$\forall x \forall y [wife(x, y) \rightarrow husband(y, x)]$$

The rule in combination with the two *facts* of the predicate `wife/2` (that need no translation) allow us to query the predicate `husband/2`. We may for example ask if it is true that Aristotle is the husband of Pythias, ask who is the husband of Pythias, ask who is the wife of Socrates, or enumerate the married couples one by one:

```
?- husband(pythias, aristotle).
true.
?- husband(pythias, Husband).
Husband = aristotle.
?- husband(Wife, socrates).
Wife = xantippa.
?- husband(Wife, Husband).
Wife = xantippa, Husband = socrates ;
Wife = pythias, Husband = aristotle.
?-
```

This simple example serves as a reminder that Prolog is not only a logic language, but also a *relational* language that can be used to check if a sentence is true, to look up the value of one argument given the value of another, or to enumerate all pairs of values. When querying a binary relation, these are the *modes* that exist.

Because Prolog can pattern-match on structured terms in clause heads, `append/3` defines list concatenation with a base case and a recursive case that peels off the head `H` of the first list and rebuilds it on the result:

```
append([], L, L).
append([H|L1], L2, [H|L]) :- append(L1, L2, L).
```

All three arguments are relational – any of them may be given or left as variables – so `?-append(Xs, Ys, Zs)` holds precisely when `Zs` is `Xs` followed by `Ys`.

Complex terms in the arguments of clauses ensures that Prolog is a *Turing complete* programming language. This is what makes it different from a language such as Datalog, which is similar in many ways, but is not Turing complete.

In order to be not only Turing complete but also a practical and efficient programming language, serious Prolog systems need to offer a lot more, and they normally do. In Section 2.1 of *Fifty Years of Prolog and Beyond*, the authors provide a conceptual and minimalist definition of the important features of Prolog, and are thus able to draw a line between what can be considered a Prolog implementation and what can not. Actually, they draw *two* lines, since they distinguish the *essential* features of Prolog from *important* (yet non-essential) features. Below, we reproduce their list of features, where 1-6 are considered essential features and 7-12 less essential. As it turns out, all the essential features except for 2) and 6) are involved when querying the binary relation between the married couples, or the ternary relation between the lists, in the examples given above.

1. Horn clauses with variables in the terms and arbitrarily nested function symbols as the basic knowledge representation means for both programs (a.k.a. knowledge bases) and queries;
2. the ability to manipulate predicates and clauses as terms, so that meta-predicates can be written as ordinary predicates;
3. SLD-resolution (Kowalski, 1974) based on Robinson's principle (1965) and Kowalski's procedural semantics (Kowalski, 1974) as the basic execution mechanism;

4. unification of arbitrary terms which may contain logic variables at any position, both during SLD-resolution steps and as an explicit mechanism (e.g., via the built-in =/2);
5. the automatic depth-first exploration of the proof tree for each logic query;
6. some control mechanism aimed at letting programmers manage the aforementioned exploration;
7. negation as failure (Clark, 1978), and other logic aspects such as disjunction or implication;
8. the possibility to alter the execution context during resolution, via ad-hoc primitives;
9. an efficient way of indexing clauses in the knowledge base, for both the read-only and read-write use cases;
10. the possibility to express definite clause grammars (DCG) and parse strings using them;
11. constraint logic programming (Jaffar and Lassez, 1987) via ad-hoc predicates or specialized rules (Fruhwirth, 2009);
12. the possibility to define custom infix, prefix, or postfix operators, with arbitrary priority and associativity.

One cannot avoid noticing that, as far as we know, no other community in support of a programming language has found itself in the position of having to determine what should be considered a *real* language of this kind. As we found in Chapter 1, the problem with Prolog is that there are so many systems around that can rightly claim to implement it but still are incompatible with each other in important ways. This is what prompted the development of a standard. For almost thirty years now, the core features of Prolog has been standardized by ISO, documented in a report known as ISO/IEC 13211-1:1995.[1]

### The logic programming landscape

Prolog is the most widely known logic programming language, but it is far from the only one. Over the past five decades, researchers have extended, restricted, or reinterpreted the Horn clause core in many directions. The following survey is necessarily selective, but it covers the formalisms and systems that will reappear in later chapters when we consider how the Prolog Web can serve as an integrating substrate for the logic programming family as a whole.

Prolog's native mode of reasoning is deduction – deriving consequences from given rules and facts – but logic programming as a field has long explored other modes of inference as well. Abductive logic programming (ALP) reverses the direction: given an observation, it searches for hypotheses that, together with background knowledge, would explain it (Kakas et al., 1992). Inductive logic programming (ILP) goes further still, learning new rules from examples and background theory (Muggle-

[1] `https://www.iso.org/obp/ui/#iso:std:iso-iec:13211:-1:ed-1:v1:en`

ton and De Raedt, 1994). Both traditions extend the reach of logic-based reasoning well beyond what a standard Prolog system provides out of the box.

Nor is deductive logic programming itself a monolith. Datalog restricts Horn clause logic to function-free programs, sacrificing Turing completeness in exchange for guaranteed termination and well-understood complexity bounds (Ceri et al., 1989; Green et al., 2013). Well-founded semantics (WFS) addresses a different limitation, providing a three-valued account of negation in which every normal logic program has a unique minimal model assigning each ground atom the value *true*, *false*, or *undefined* (Van Gelder et al., 1991); systems such as XSB and SWI-Prolog implement WFS via tabling, giving them stronger semantic guarantees than standard Negation As Failure. Answer Set Programming (ASP) departs more radically, searching for the stable models of a logic program rather than computing answers by resolution; a recent variant, s(CASP) (Arias et al., 2018), combines ASP with constraint logic programming in a top-down, query-driven execution model with support for explanation generation. Probabilistic logic programming (PLP) equips logic programs with probabilistic reasoning: ProbLog (De Raedt et al., 2007) is a Python-based toolbox that defines distributions over possible worlds and supports inference, sampling, and parameter learning, while cplint (Riguzzi, 2018) is a complementary SWI-Prolog library for LPADs under the distribution semantics, accessible both locally and through "cplint on SWISH." Constraint logic programming (CLP) extends Prolog's unification with constraint solvers over domains such as finite integers, reals, and sets, and most major Prolog systems now ship with CLP libraries as standard. Finally, Logic Production Systems (LPS) (Kowalski and Sadri, 2015) combines logic programming with reactive, event-driven computation.

The common thread is that logic programming is not a single language but a family of formalisms. This breadth will become relevant in later chapters, where we show how the Prolog Web can accommodate any system in this family that can accept a goal and return answers.

## 2.2 Web Prolog in a nutshell

Web Prolog is designed as a superset of a subset of the ISO Prolog core standard. The dialect defined by ISO is well understood, and is reasonably close to most of the dialects used in Prolog textbooks. The ISO Prolog syntax specification as well as a large subset of its built-in predicates form an excellent point of departure for our attempt to create a design and a standard for Web Prolog.

The syntax of Web Prolog is exactly as in ISO Prolog, except that three infix operators (`!/2`, `if/2` and `@/2`) that are not in the standard are defined. They can easily be added by means of `op/3`, the ISO Prolog predicate allowing a programmer to define prefix, postfix or infix operators and specify their precedences. This means that syntactically well-formed Web Prolog programs can always be *read* (but not necessarily *run*) by a conforming implementation of ISO/IEC 13211-1:1995. It is easy to imagine circumstances where this might come in handy.

The capabilities of a programming language are to a large extent determined by the built-in constructs it provides. Rather than to make a long list, Figure 2.1 shows a "cloud" of built-in control constructs and predicates specified by the ISO standard.

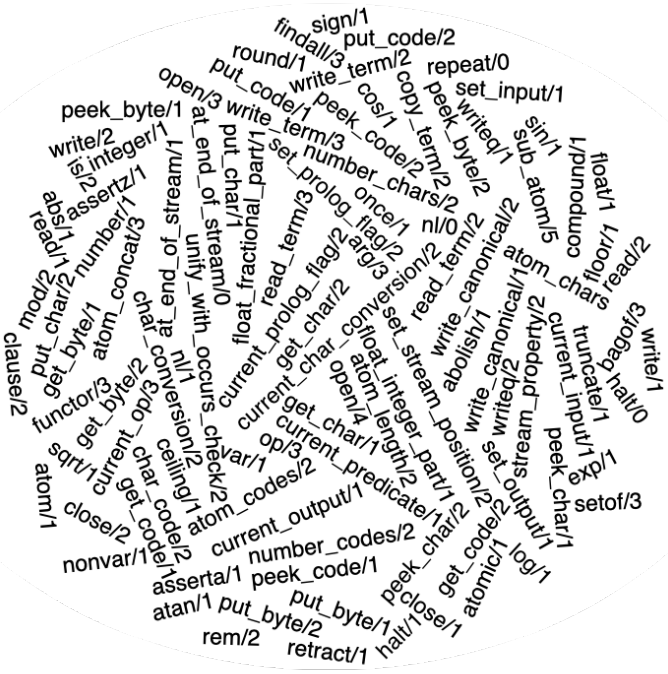



**Fig. 2.1** A cloud of ISO Prolog built-in predicates.

Most of the predicates in this cloud *can* and *should* be supported by Web Prolog. (We will get to the exceptions a bit further ahead.) However, while most of them are necessary, they are not sufficient. Here is what else we think is needed:

- In addition to the built-in predicates offered by the ISO Prolog core, Web Prolog needs to support a set of *standard library predicates* that conforming nodes on the Prolog Web must normally offer. They should just be there – a programmer should not have to load them explicitly. The *Prologue for Prolog* lists `member/2`, `append/3`, `length/2`, `between/3`, `select/3`, `succ/2`, `maplist/2-8`, `nth0/3`, `nth1/3`, `nth0/4`, `nth1/4`, `call_nth/2`, `foldl/4-6` and `countall/2`.[2] See also the public-domain set of libraries developed by the Prolog Commons Working Group.[3]
- Item number 10 in the list of essential and important features points to the possibility to express Definite Clause Grammars (DCGs) and parse strings using

[2] `https://www.complang.tuwien.ac.at/ulrich/iso-prolog/prologue`

[3] `http://prolog-commons.org`

them. DCG notation and the associated parsing primitives are specified separately as an extension, in ISO/IEC 13211-3 (Part 3: Definite clause grammar rules).
- As we are aiming for a profile of Prolog suitable for programming the Web we need predicates such as `http_open/3` for accessing its resources. Also, JSON serialization/deserialization seems essential to have.
- Last but not least, we extend our subset of ISO Prolog with carefully crafted concurrency and distribution primitives heavily inspired by Erlang. When we write "heavily inspired," we really mean it. We try to stay as close to Erlang as possible, we use the same *kind* of actor as Erlang, and we use the same names for our concurrency-oriented predicates that Erlang uses for the corresponding functions. This is not to say that we never deviate from the way Erlang does its thing, because as we shall see, we do when we have good reason to.

We would expect the community to be able to agree on the first three points, and the last point in the list is likely to present the only major technical challenge. It is not *much* of a challenge however, since Erlang shows us where we are lacking, and where we need to go.

There are features, built-in predicates and libraries that may never make it into Web Prolog. The reason why may vary – they may be dangerous, or they may be superfluous. Other features may be deemed not developed enough to be included at this point in time.

- We only use a subset of ISO Prolog as it contains primitives that do not seem to "belong" on the Web and which may also compromise the security of a node, such as predicates that gives a program access to the operative system.
- Even though they are certainly relevant to the Web, there is definitely no need for predicates for building web servers, such as the ones available in SWI-Prolog's `library(http/http_server)`. After all, a Prolog node *is* a web server.
- Even though predicates for constraint logic programming is listed as an important Prolog feature, and unless they are already defined by ISO, we do not believe time is ripe to include them in Web Prolog. As they mature, this may of course change.
- We expect that it is possible to make only limited use of modules. After all, a node *is* a module, and an actor may be seen as a module too.

We have characterized Web Prolog as a profile of Prolog suitable for programming on the Web. As we shall see, Web Prolog can be cut up into subsets that are themselves profiles, or sub-profiles if you will. Some such profiles of Web Prolog do not support predicates for database updates such as assert and retract, or predicates for reading from and writing to a terminal. Other profiles do not support `op/3`.

FYPB argues that the ISO core standard helped establish a common kernel and improved developer confidence in portability, but also that portability problems persist and that converging on shared libraries and tools remains difficult in practice. From that perspective, Web Prolog is best seen as a portability strategy rather than a language fork: we keep the ISO syntax, constrain dangerous or non-web-centric

features, and then standardise a small, web-relevant set of libraries and concurrency primitives that every node is expected to provide. This shifts the portability problem from "porting whole systems" to "agreeing on profiles and their mandatory capabilities" – a more tractable target in an open ecosystem.

In the terminology of the paper, this also aligns with more agile convergence efforts (such as Prolog Commons) that aim to improve practical compatibility and shared infrastructure without requiring the full weight of an ISO revision process.

### 2.2.1 Web Prolog is inspired by Erlang

> I would prefer multi-threading in Prolog to look as much as possible like Erlang.
>
> *Richard O'Keefe*[4]

Erlang is a general-purpose, concurrent, functional programming language developed by Joe Armstrong, Robert Virding and Mike Williams in 1986. Regarded as the first actor programming language to gain popularity, it started out as a proprietary language within Ericsson,[5] but was released as open source in 1998.[6] Erlang was designed with the aim of improving the development of telephony applications, but has in recent years, thanks to its built-in support for massive concurrency, distribution and fault tolerance, been used as a very capable language for implementing the server-sides of web applications,[7] as well as for programming a wide range of soft real-time control problems (Virding et al., 1996).

Inspired by the list of essential features of Prolog given in the previous section, and if given the task of formulating a similar list defining what it means to be an Erlang-style programming language, we would likely include at least the following four essential requirements:

1. The ability to execute a large number of actor processes concurrently;
2. the ability of actors to keep their states isolated so that no one else can mutate them directly;
3. the ability to use message-passing for asynchronous communication between actor processes, avoiding mutable shared memory and locking issues;
4. the ability to support network-transparent concurrent programming where actors are allowed to create other actors on remote computers and enter into seamless communication with them.

---

[4] `https://groups.google.com/g/erlang-programming/c/1jdsnqZ4XfQ/m/ve9WfFl2YBwJ`

[5] `https://www.ericsson.com`

[6] A good overview of the Erlang language, its design intent and the underlying philosophy, is given in the Ph.D. thesis of Joe Armstrong (Armstrong, 2003b).

[7] By companies such as WhatsApp and Klarna for example.

A more succinct expression that captures the essence of Erlang is to characterize it as a language for *message-passing concurrency and distribution*.

These are four essential features that if added to the features of ISO Prolog will provide us with the most crucial parts of a foundation for Web Prolog, and indeed for the whole Prolog Trinity ecosystem. Of course, for this to work we must make sure that the two sets of features are compatible. Our current understanding is that they are, and we intend to demonstrate this in the book.

Why did we choose Erlang as a source of inspiration, rather than any alternative? After all, there are other actor programming languages – Akka, Pony and E, to name a few. The main reason for the choice of Erlang is that it is arguably the most *mature* concurrency-oriented language in existence, with a bigger community than the alternatives, and a community that includes a fair number of industrial users. Many millions of lines of source code have been written in Erlang, many books teaching the language have been authored, and university courses teaching concurrent and distributed programming often uses Erlang. In our opinion, since we are aiming for a standard it makes a lot of sense to borrow the required concurrency-oriented features from a language as mature and battle-tested as Erlang.

Another reason for our choice is that Erlang and Prolog are in many ways remarkably similar, both when it comes to syntax, and the reliance on dynamic typing, assign-once variables, pattern matching and recursion. The similarities can be explained by the fact that historically, Erlang evolved from Prolog and the first version of Erlang was actually implemented as an interpreter in Prolog (Armstrong et al., 1995). The following listings of Erlang code (to the left) and Web Prolog code (to the right) show some of the similarities as well as some of the differences between the two languages:

```
append([], L) -> L ;
append([H|L1], L2) ->
    [H|append(L1, L2)].
```

```
append([], L, L).
append([H|L1], L2, [H|L]) :-
    append(L1, L2, L).
```

Note the use of capital letters for variables, the familiar notation for lists, and the reliance on pattern matching and recursion. A major difference is that Erlang is a functional programming language, whereas Web Prolog is relational. Since a function is just a special case of a relation this difference must not be exaggerated, but it does show in the way function calls may be nested, something that cannot be done in Prolog (or Web Prolog) since arguments are used to represent outputs as well as inputs. What *can* be done in Prolog, however, but not in Erlang, is to call `append/3` in more than one *mode* – not only to append one list to another, but also, for example, to nondeterministically split a list up into two parts. Furthermore, thanks to logic variables, `append/3` is tail-recursive in Prolog, but not in Erlang's `append/2` (as it is written here).

Despite such differences, as long as we do deterministic computation only, and no search is involved, logic programming and functional programming are fairly similar in the way they work, and methods used to achieve success with one often transpose

to the other. Again, features such as pattern matching, assign-once variables and recursion play important roles in both languages.

During its evolution, Erlang gradually shed several features that Prolog had – and still has – while at the same time developing exceptionally strong support for concurrent and distributed programming, an area where Prolog historically lacked, and still lacks, comparable built-in capability. Key here is the concept of an actor, and the ways actors can be scripted. Below, the echo server written in Web Prolog is placed side-by-side with an implementation in Erlang – a predicate in Web Prolog and a function in Erlang – both making a recursive call that implements a loop:

```
echo_actor() ->
    receive
        {echo, From, Msg} ->
            From ! {echo,Msg},
            echo_actor()
    end.
```

```
echo_actor  :-
    receive({
        echo(From, Msg) ->
            From ! echo(Msg),
            echo_actor
    }).
```

The Web Prolog code on the left demonstrates the use of two constructs foreign to traditional Prolog, implementing the sending and receiving of messages in the style of Erlang. The programs have a similar look, and this is intentional. We have *strived* to make Web Prolog look as similar as possible to Erlang within the constraints imposed by the syntax and semantics of ISO Prolog.[8]

A function call such as in `Pid = spawn(fun() -> echo_actor() end)` (or, but only for zero-argument functions, `spawn(fun echo_actor/0)`) can be made in order to spawn our echo server in Erlang, while doing it in Web Prolog would use something like `spawn(echo_actor, Pid)`. These calls look somewhat similar too, but while Erlang is a higher-order language in which the spawn function takes an anonymous function as its argument, Prolog (or Web Prolog) is not a higher-order language in this sense. In Web Prolog, `spawn/2` is a *meta predicate* which expects a callable term to be passed in the first argument, a term that when called will execute a procedure that determines how the actor will behave.

An important observation in FYPB is that when portability problems are not solved but merely worked around, the community tends to fragment into smaller, incompatible subgroups, making shared tooling and shared libraries harder to sustain. This matters here because concurrency and distribution are precisely the kinds of features that, if left to ad-hoc, system-specific designs, will amplify dialect drift. One motivation for staying close to Erlang – even down to predicate names and a small set of primitives – is therefore to reduce the degrees of freedom. If we are going to add distributed concurrency to Prolog, it is better to do so in a way that maximises the chance of convergent implementations.

[8] Note that `!/2` here is the Erlang-style *send* operator, rather than the Prolog cut (`!/0`).

### 2.2.2 Web Prolog is a multi-paradigm programming language

> Prolog is different, but not that different.
>
> *Richard O'Keefe*

A common way to categorize programming languages is in terms of the programming paradigm(s) they support. In the very nicely organized and very well argued taxonomy of programming paradigms by Peter van Roy in (van Roy, 2009), depicted in Figure 2.2,[9] it is indeed possible to pinpoint exactly where Web Prolog fits in. We have enclosed the relevant square boxes in boxes with rounded corners.

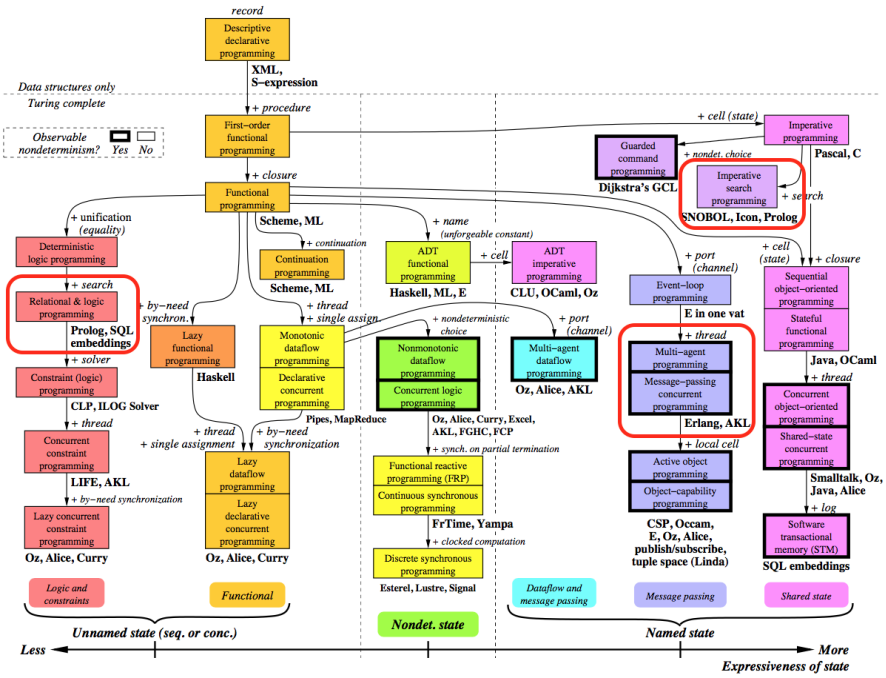



**Fig. 2.2** A taxonomy of programming paradigms due to Peter van Roy. (Image source and license to be confirmed.)

This suggests that we can regard Web Prolog as a language that supports three programming models: *relational logic programming* (like in Prolog), *imperative programming* (also like in Prolog), and (like in Erlang) *message-passing concurrent programming*. (A note on terminology is in order. In this book we prefer the term *actor-based programming*. Joe Armstrong used to refer to it as *concurrency-oriented programming*, or COP.).

[9] The diagram is available at https://www.info.ucl.ac.be/~pvr/paradigmsDIAGRAMeng108.pdf

With access to predicates for asserting and retracting clauses in the dynamic database, traditional Prolog has always, albeit somewhat reluctantly, been able to support imperative programming (and later additions such as `setarg/3` has of course amplified this). Also, the built-in backtracking *search* that is so useful in Prolog has little to do with logic. Thus Prolog has never really been a single-paradigm programming language, and this is reflected in van Roy's taxonomy. In (van Roy, 2009) he suggests that this is not a bad thing:

> A good language for large programs must support several paradigms. One approach that works surprisingly well is the *dual-paradigm language*: a language that supports one paradigm for programming in the small and another for programming in the large.

With the addition of support for the actor programming model, Web Prolog becomes a much clearer as well as (in our view) a more interesting case of a multi-paradigm programming language. In particular, it can be argued that the support for programming in the large has been greatly improved.

### A sip of Elixir and the rise of Phoenix

> The rise of popularity of the Internet and the need for non-interrupted availability of services has extended the class of problems that Erlang can solve.
>
> *Joe Armstrong*

Erlang has successfully been used on the server sides of many web applications. However, this is also an area of application which in recent years has been taken over by the Elixir programming language.[10] A program written in Elixir compiles into the Erlang VM and therefore belongs in the same paradigm and shares the same abstractions for building concurrent, distributed and fault-tolerant applications. The main difference concerns syntax.

The syntax of Elixir is inspired by the Ruby programming language. Here is how the programs in Section **??** looks like when written in Elixir:

```
# length

def length([]), do: 0
def length([_h|t]), do: 1 + length(t)

# naive reverse

def reverse([]), do: []
def reverse([h|t]), do: reverse(t) ++ [h]

# spawning example
```

[10] `https://elixir-lang.org/`

```
self = self()
spawn fn -> send(self, length([:a,:b,:c])) end
receive do
    m -> IO.puts(m)
end
```

The Phoenix web framework[11] is an add-on to Elixir which is inspired by Ruby on Rails,[12] a server-side MVC-based web application framework written in Ruby that was very popular only a decade ago and which can be said to have set the standard for such frameworks. In addition, and more importantly, Phoenix takes advantage of the Erlang VM's ability to handle millions of processes and connections in order to offer bidirectional real time capabilities with channels between JavaScript on the client and Elixir on any of the servers in a cluster. Each single visitor to a website can have its own process on the server and its own real time connection. This opens up possibilities that are not present in traditional web frameworks.

Note that it is true also for the Prolog Web that each client can have its own process on the server (i.e. on the node) and its own real time connection. It will not scale as well as with Elixir+Phoenix, but this might improve in the future.

Finally, keep in mind that the Prolog Web is not a framework, but an extension of the Web. Elixir+Phoenix is not webized in the same strong sense as Web Prolog and the Prolog Web, and its creators do not aim for it to become an official web standard.

### Is Web Prolog an object-oriented language?

Object-oriented features are often presented as a practical route to programming in the large. In *Fifty Years of Prolog and Beyond*, the lack of object orientation is identified as one of Prolog's weaknesses, and the integration of logic programming with object-oriented features is described as an "elusive goal." Nevertheless, the authors present a number of what they call "effective" proposals for object-oriented extensions to Prolog, developed by various researchers and system builders. Among these, Paolo Moura's *Logtalk* stands out as particularly influential.

Interestingly, Alan Kay – often credited with inventing object-oriented programming (OOP) – seems to regard message passing as the fundamental concept in object-oriented programming, rather than the more visible constructs of classes and objects. He even considers *Erlang* itself to be an object-oriented language:[13]

> Significant parts of Erlang are more like a real OOP language than the current Smalltalk, and certainly the C based languages that have been painted with "OOP paint."

What is evident is that *if* Erlang qualifies as an object-oriented programming language, then Web Prolog too must be considered one. And *if* so, this means that

[11] https://phoenixframework.org

[12] https://en.wikipedia.org/wiki/Ruby_on_Rails

[13] https://computinged.wordpress.com/2010/09/11/moti-asks-objects-never-well-hardly-ever/

Web Prolog should probably be added to the list of object-oriented extensions of Prolog referenced by the authors of FYPB. But those are *big* ifs, and in a blog post Armstrong writes:[14]

> As Erlang became popular we were often asked "Is Erlang OO" – well, of course the true answer was "No of course not" – but we didn't to *[sic]* say this out loud – so we invented a serious of ingenious ways of answering the question that were designed to give the impression that Erlang was (sort of) OO (If you waved your hands a lot) but not really (If you listened to what we actually said, and read the small print carefully).

So we still prefer to refer to Web Prolog processes as "actors" and "agents," rather than "objects," and to frame the paradigm as "actor-programming" or "agent-oriented programming" rather than "object-oriented programming." Programmers looking for an OO-system that is class-based and supports inheritance would likely prefer Moura's Logtalk, while the rest of us may be happy with the features provided by Web Prolog.

### Any unexpected interactions between paradigms?

It is probably wise to entertain a suspicion of unexpected interactions between language features and possible impedance mismatches between the two paradigms – between Prolog's relational, nondeterministic programming model based on logic and Erlang's functional and message passing model. How well do the Erlang-style constructs mix with Prolog – with backtracking for example, or with the features for imperative programming? What do we get if we combine them? A kludge, or something quite beautiful? (This, as so many other things, might be in the eye of the beholder, but we know what *our* eyes tell us.)

In theory, we should be on the safe side. Sequential Erlang is basically Erlang with its data types, one-way pattern matching, functions and control structures, but without spawn, send, receive, and other constructs used for concurrent programming. The idea behind Web Prolog can thus be described as an attempt to "plug out" the sequential part from Erlang and "plug in" sequential Prolog instead. There seems to be no principled reasons why we would not be able to replace the sequential functional language with a sequential relational language, e.g. a logic programming language such as Prolog. Indeed, Joe Armstrong (2003a) writes:

> Erlang is a concurrent programming language with a functional core. By this we mean that the most important property of the language is that it is concurrent and that secondly, the sequential part of the language is a functional programming language.
>
> The sequential subset of the language expresses what happens from the point it time where a process receives a message to the point in time when it emits a message. From the point of view of an external observer two systems are indistinguishable if they obey the principle of observational equivalence. *From this point of view, it does not matter what family of programming language is used to perform sequential computation.* (Our emphasis.)

Alternatively – and this has been our choice – the approach can be described as an attempt to *extend* Prolog with constructs such as spawn, send and receive. The idea is

[14] `https://harmful.cat-v.org/software/OO_programming/why_oo_sucks`

to keep everything that core Prolog has to offer, and extend it with a number of those primitives that make Erlang such a great language for programming message-passing concurrency. The choice between extending Prolog with Erlang-style constructs and extending Erlang with Prolog-style constructs is easy to make, and a lot has to do with syntax. Provided we can accept using a syntax which is relational rather than functional, precluding the nesting of function calls, it may be clear already at this point that the surface syntax of Prolog can easily be adapted to express the needed Erlang-style primitives. It is arguably a lot harder to express Prolog rules and other constructs using the syntax of Erlang.

### 2.2.3 Web Prolog as a scripting language for the Web

> Scripting languages are a lot like obscenity. I can't define it, but I'll know it when I see it.
>
> *Larry Wall*

Over the years, the Web has become a key delivery platform for a variety of sophisticated interactive applications in just about any conceivable domain. As a consequence, JavaScript, more or less the only game in town for programming the front-end of a web application, has become among the most commonly used programming languages on Earth. Indeed, JavaScript must be regarded as *the* web programming language of our times.

JavaScript started out as a very small, highly domain-specific scripting language that was limited to running within a web browser to dynamically modify the web page being shown, but has over time evolved into a widely portable general-purpose programming language. Modern JavaScript is a high-level, dynamically typed, interpreted multi-paradigm programming language with several essential features that make it a versatile tool for web development. JavaScript's syntax supports various programming paradigms, including imperative, functional, and object-oriented programming styles. With features such as callbacks, promises, and async/await, JavaScript supports asynchronous programming, enabling non-blocking operations, especially useful in web applications. JavaScript's native format for data interchange, JSON, is lightweight and widely used for data transmission. JavaScript runs on almost all modern web browsers and platforms, making it universally applicable for web development. It easily interfaces with numerous web APIs for tasks like accessing the Document Object Model (DOM), making HTTP requests, and handling events. This makes it an excellent tool for connecting disparate systems in web applications.

With the advent of Node.js,[15] JavaScript extended its reach to server-side development. This allowed for a unified language experience across both client and server sides, simplifying the development process by allowing the same language to be used throughout the entire stack. Not having to learn more than one language in order to become a so called "full-stack developer," and not having to switch languages when

[15] `https://en.wikipedia.org/wiki/Node.js`

changing from working on the server-side to working on the client-side are important advantages. Also, Node.js is frequently used to build and connect microservices. Its non-blocking I/O model and event-driven architecture make it suitable for building scalable and efficient network applications.

It is clear that as a relational logic programming language with the essential and important features listed above, Prolog is very different from JavaScript in terms of paradigms supported. The addition of Erlang-style concurrency and asynchronous communication does not really narrow the gap, as those are features that work differently in Erlang. What matters here is the *role* that JavaScript plays on the Web. We want Web Prolog to play a similar role in the Prolog Trinity ecosystem as JavaScript does in the JavaScript ecosystem, just as clear-cut.

The connection between scripting languages and the Web has been noted before. For example, in his article "Scripting: higher level programming for the 21st Century," John Ousterhout (1998) writes (about the Internet rather than the Web, but that makes little difference):

> The growth of the Internet has also popularized scripting languages. The Internet is nothing more than a gluing tool. It does not create any new computations or data; it simply makes a huge number of existing things easily accessible. The ideal language for most Internet programming tasks is one that makes it possible for all the connected components to work together; that is, a scripting language.

So what about the suitability of Web Prolog for programming the Web? Can Prolog challenge JavaScript in this space, at least for some applications, such as web-based AI? For this to be possible, we must ensure that Web Prolog is

- viable as a scripting language,
- suitable for programming the Web,
- possible to implement in browsers, and
- secure.

### Web Prolog as a scripting language

As regards the viability of Web Prolog as a scripting language we note that while hardly a definition, Ousterhout (1998) characterizes scripting languages as follows:

1. They are suitable for "programming in the large,"
2. they are designed for "glueing" applications together,
3. they tend to be typeless, thus making it easier to connect components,
4. they are usually interpreted, thus providing rapid turnaround during development by eliminating compile times,
5. they are higher level than a system programming language in the sense that a single statement does more work on average, and
6. they allow rapid prototyping.

JavaScript does indeed fit this characterization, but given the development described above, it should now probably be described as general-purpose languages which also

happens to be suitable for scripting. By contrast, languages such as C++ and Java are not well suited to that purpose..

What about Prolog? Does it satisfy Ousterhout's characterization? The points 3-6 in his list are probably true of the Prolog programming language in general. (With the possible exception of 4 – some systems compile, but do it very quickly.) Whether or not the points 1 and 2 applies varies between systems. For example, the people behind the commercial SICStus Prolog appears to regard Prolog more as a language for writing problem-solving modules to be embedded into other code, written in Java, C++, Python or what have you.[16] It is only a guess, but we suspect that the people behind SICStus Prolog would not be willing to characterise it as a scripting language, although that may simply be a business decision more than anything else. In contrast, acting as a glue language and a language for programming in the large is the ambition of SWI-Prolog, witness the following quote from the online manual:[17]

> SWI-Prolog positions itself primarily as a Prolog environment for 'programming in the large' and use cases where it plays a central role in an application, i.e., where it acts as 'glue' between components.

Interestingly, the people behind Erlang also think of it as a glue language:

> We use Erlang as the glue to handle all orchestration, and then we use Python, C, Julia, ... It is actually a language *intended* to act as a hub towards other languages. The interfaces could be protocols, could be RESTful APIs, or other programming languages. It's ideal for that.[18]

Erlang-style, network-transparent message-passing concurrency seems ideal for programming in the large on the Web, and for orchestrating both local and remote processes. Erlang glue is likely a different kind of glue, with different properties, than Prolog glue. There is hope that combining Prolog and Erlang might yield an even stronger and more flexible glue of the required kind – a *superglue*, if you will.

Despite the good fit with Ousterhout's characterization, neither Prolog nor Erlang are usually *advertised* as scripting languages, and they were hardly *designed* for the purpose of glueing applications together (or at least Prolog was not). They are both full-blown very flexible general-purpose programming languages with "batteries included." Like JavaScript, they should probably be regarded as general-purpose languages which also *happens* to be suitable for scripting.

Web Prolog is a very *general* special-purpose scripting language. Similar to most other scripting languages, Web Prolog can be used for purposes for which "scripting" is not really the right word. Web Prolog comes with a focus on web *logic* programming as well as web *agent* programming, and in client-side browsers as well as server-side. Using Web Prolog, the owner of a node is for example able to build knowledge bases consisting of many millions of clauses – facts as well as rules, and/or web agents that can make use of such knowledge bases.

---

[16] Mats Carlsson, main developer of SICStus Prolog, personal communication

[17] `http://www.swi-prolog.org/pldoc/man?section=swiprolog`

[18] See `https://www.youtube.com/watch?v=K8nxTSPHZhs5`, 8:58 into the discussion.

### Is Web Prolog suitable for programming the Web?

To demonstrate that Web Prolog is suitable for programming the Web is what this book is about. But we are going further than that, as we are also aiming for a novel extension of the Web – the Prolog Web and even an Agentic Web – and a complete ecosystem, for which Web Prolog will serve as the default language.

### Web Prolog running in browsers

Regarding our third point, for Web Prolog to become a viable web programming complement to JavaScript it is vital that it can be implemented in browsers too, since this comes with a number of advantages:

1. When network connection is slow, it is best to perform the majority of computations in the browser.
2. This is where, in most scenarios, the *state* of an interaction between a user and an application should preferably be represented.
3. Client-side computation reduces the demands placed on nodes.
4. This is also where we find browser APIs that lets a web developer manipulate the DOM, store data locally, and add features such as spoken interaction, video or graphics to an application.

For Web Prolog in the browser to play the same role as JavaScript in the browser currently does, it must allow Web Prolog source code to be loaded from any server on the Web by means of the HTML `<script>` element, inline or with a link.

We do not, however, go all the way in our attempt to replicate Javascript's abilities as a scripting language. One thing that we do not at this time aim for is to make Web Prolog into a language with primitives for scripting the Document Object Model (DOM) in web browsers. With time it may well be interesting to develop such capabilities, but we believe it is simply too early to try to standardize them. In fact, we suspect that JavaScript will reign supreme in that role, and that if a browser is running Web Prolog locally, it will be as a *web worker*,[19] with which the main JavaScript process is talking.

### Web Prolog and security

As for security, similar to JavaScript, Web Prolog is a *sandboxed language*, open to the execution of untested or untrusted source code, possibly from unverified or untrusted clients without risking harm to the host machine or operating system. Therefore, Web Prolog does not include predicates for file I/O, socket programming or persistent storage, but must rely on the host environment in which it is embedded for such features.

---

[19] `https://en.wikipedia.org/wiki/Web_worker`

Web Prolog does indeed share some of the properties that made JavaScript succeed on the Web. A visit to a web application server starts a JavaScript process on the client, running code that has been downloaded from the application's host to the user's client. Such a process must be allowed to run there (i.e. the owner of the client must allow it), and when it is (and most users allow it by default), it must execute in a way that does not harm the client. When a Web Prolog process is run on a node it must be allowed by its owner to do so, and it must run without harming the node. So one thing they share, Web Prolog and JavaScript, is the ability to run untested and untrusted source code, authored by unverified and untrusted programmers, in a sandbox on someone else's computer.

In summary, while JavaScript's initial role was confined to client-side scripting in web browsers, its capabilities have expanded significantly. Today, it serves as a versatile glue language that connects various components in both client and server environments, as well as in the broader web development ecosystem. When designing Web Prolog and the rest of the Prolog Trinity ecosystem, we must of course try to learn from such a popular and versatile tool for web development, with a focus on the good parts in particular.

## 2.3 Erlang-style message-passing concurrency in Web Prolog

> In a nutshell, if you were an actor in Erlang's world, you would be a lonely person, sitting in a dark room with no window, waiting by your mailbox to get a message. Once you get a message, you react to it in a specific way: you pay the bills when receiving them, you respond to Birthday cards with a "Thank you" letter and you ignore the letters you can't understand.
>
> *Fred Hébert*

Corresponding to the send, receive and spawn operations there are built-in Web Prolog predicates such as `!/2`, `receive/1-2` and `spawn/1-3`. In this section, the use of these and other predicates will be demonstrated by examples. We choose to work with very simple examples, many of which are borrowed from tutorials and text books teaching Erlang to beginners. The major motivation for having it in this way is to try to satisfy two kinds of readers who might want to approach the material in two different ways. Readers who have a year or two of Prolog programming under their belt, but feel that time is ripe to have a look at concurrent programming, may want to look at the examples very carefully and perhaps work through them themselves using the Trinity Prolog implementation.

Readers who are very experienced Prolog programmers and teachers may also want to ask themselves if this a good way to teach students of Prolog some concurrent programming techniques, or if something else might work better. Their role is as evaluators of potential teaching material rather than learners. We know what we believe; we think Web Prolog might be suitable for teaching both Prolog-style logic programming and Erlang-style actor programming in the same system, and preferably in a web-based playground – a great way to create as little hassle for a teacher as possible.

Then, of course, we expect the latter kind of reader to evaluate our proposal and to determine if Erlang-style concurrent programming makes sense in the context of Prolog. (We know what we think – it does.)

### 2.3.1 Programmer talking to actor, actor talking to itself

If we need to hold a Prolog-style conversation with an actor, we may want to talk to a Prolog *shell*. By using a terminal attached to a shell, we can interact with it the way we normally interact with Prolog, by querying it, updating its dynamic database, and so on. For example, here is how we instruct the shell to split a list up into a prefix and a postfix using `append/3`:

```
?- append(Xs, Ys, [a,b,c]).
Xs = [], Ys = [a, b, c] ;
Xs = [a], Ys = [b, c] ;
Xs = [a, b], Ys = [c] ;
Xs = [a, b, c], Ys = [] ;
false.
?-
```

This book, as we have warned, is not a beginner's textbook, so how Prolog is able to come up with solutions to such queries is not something we will cover. Our focus is rather on the concurrency-oriented, message-passing features of Web Prolog, so we begin instead by introducing a couple of simple messaging primitives. But first, using `self/1`, we need to determine the identity of the shell we will be talking to:

```
?- self(Self).
Self = 85234512.
?-
```

What we got back here is a *pid*, an unforgable and locally (but not necessarily globally) unique identifier. Our proposal here is to use random integers of a size that makes it impossible in practice for anyone to *guess* the pid of an actor.

As stated above, if we know the pid of an actor process we can send it a message. Just as any other kind of actor, the shell is equipped with a mailbox, and this is where the message will end up. The syntax and semantics of the send operator is easy to explain. With a call such as `Actor ! Message` the term `Message` is sent to the mailbox of the process identified as `Actor`, either by means of a pid or (as we shall see) using a registred *name*. For example, using `!/2` with the pid of our shell we can instruct it to send *itself* a message:

```
?- 85234512 ! hello.
true.
?-
```

Now we can use `receive/1` to make sure that the message we sent was received by the shell and is present in its mailbox. A single receive clause with a variable in the head and `true` in the body does the job:

```
?- receive({Message -> true}).
Message = hello.
?-
```

This is one of the simplest use of the receive operation we can think of, but `receive/1` really is more complex than this, so expect more to come further ahead in this chapter. One more thing about the above call to `receive/1` is worth a mention already at this point though: had the mailbox been empty, `receive/1` would have *blocked* the execution of the process running as a shell until a message shows up.

There is an even simpler call to `receive/1` that one can make, namely this:

```
?- receive({}).
```

This call does not wait for a any specific message, which just means that it will block forever, or until the the process that runs it is killed. Control-C will work and send the user back to the `?-` prompt.

### 2.3.2 Two utility tools for programming in the shell

Copying and pasting pids is not very convenient. Instead, we can use a small shell feature (borrowed from SWI-Prolog): if a variable name is prefixed with a dollar sign, the shell substitutes it with the variable's most recent binding before running the query. For example:

```
?- $Self ! hello.
true.
?-
```

Here, the shell kept track of the most recent binding of `Self` to a value (`85234512` in this case) and substituted the term `$Self` with that value before the query was run.

During interactive development, it is also important to inspect the shell's mailbox. Using `receive/1` is often awkward here: if the mailbox happens to be empty, `receive/1` will block, effectively freezing the session until the call is interrupted (e.g. with Control-C). Moreover, if we want to examine *all* pending messages, we would need to call `receive/1` repeatedly, without necessarily knowing how many messages there are – which again risks blocking, or that messages are left behind.

A more convenient tool is the utility predicate `flush/0`. It prints (and removes) all messages currently in the shell's mailbox and, crucially, it never blocks.

Below, we demonstrate both of these utilities by first sending yet another message to our shell and then use `flush/0` to inspect and empty the contents of its mailbox:

```
?- $Self ! goodbye.
true.
?- flush.
Shell got hello
Shell got goodbye
true.
?-
```

The $Var substitution mechanism (or "dollar notation") in combination with `flush/0` comes in very handy during interactive programming in the shell and we shall rely on them extensively when running examples through all of the book.

### 2.3.3 Actors talking to other actors

It takes two to tango, and an actor talking to itself is not very interesting. Fortunately, as already mentioned, an actor is capable of spawning other actors and then start talking to them.

The built-in predicate `spawn/1-3` is used to create new actor processes. In a call such as `spawn(Goal,Pid)` it expects a callable goal to be passed in the first argument, and will bind the variable in the second argument to the pid of the actor created. Here is how this might look like:

```
?- spawn(echo_actor, Pid).
Pid = 72347585.
?-
```

The goal calls a predicate which determines the behavior of the actor. The actor process runs in parallel with the caller, the shell in this case. Here is an example script that implements a simple echo server:

```
echo_actor :-
    receive({
        echo(From, Msg) ->
            From ! echo(Msg),
            echo_actor
    }).
```

We have seen this piece of code before, and it is time to explain how it works. The predicate `echo_actor/0` defines a loop where a call to `receive/1` tries to select a message of the form `echo(From,Msg)` from the mailbox and send the echo message back to the actor referenced by the pid to which the variable `From` is bound, and then continue looping by making a recursive call.

To make our server return the echo message to the shell we must send it a message of the form `echo(Pid,Msg)` with `From` bound to the pid of the shell and `Msg` bound to the message. Here is how we do that, this time supported by our shell utilities:

```
?- self(Self), $Pid ! echo(Self, hi).
Self = 85234512.
?- flush.
Shell got echo(hi)
true.
?-
```

The sending is asynchronous, i.e. `!/2` does not block waiting for a response but continues immediately. Also, sending a message to a pid always succeeds and never throws an error, even if the pid points to a non-existing process. In that case the message is simply discarded. This means that short of having the targeted actor return a confirmation message we will not always know if the message reached it. However, the above case did not leave us in the dark, as the actual echo served as a confirmation message.

Web Prolog guarantees per-sender FIFO delivery: for any fixed sender–receiver pair, messages are enqueued in the receiver's mailbox in the order they were sent. There is no single total order across different senders; concurrent sends to the same receiver may be observed in either order by `receive/1-2`.

### Monitoring

In a call such as `spawn(Goal, Pid, Options)` the optional third argument is a list of options used for configuring the actor. One such option is `monitor`, which if set to `true` requests that the parent process be notified when the spawned actor terminates.

In the following example, `monitor(true)` instructs the child actor to send a `down` message to its parent just before terminating:

```
?- spawn(2 > 1, Pid, [
       monitor(true)
   ]).
Pid = 60367387.
?- flush.
Shell got down(60367387,60367387,true)
true.
?-
```

A `down` message has the form `down(Ref, Pid, Reason)` where `Pid` is the process identifier of the terminated actor, `Reason` describes *why* it terminated, and `Ref` identifies the *monitor instance* that produced this notification. When monitoring is installed via `monitor(true)`, the monitor reference is chosen to be identical to the child's pid, which is why the first two arguments coincide in the example.

In addition to the `monitor` option, the predicate `monitor(Pid, Ref)` is provided, which installs a monitor and returns a fresh reference in `Ref`. The reference matters whenever there may be more than one monitor, because each monitor produces its own `down` message and we must be able to tell them apart:

```
?- spawn(sleep(3), Pid),
   monitor(Pid, Ref1),
   monitor(Pid, Ref2).
Pid = 41544343,
Ref1 = 10924110,
Ref2 = 92041933.
?- flush.
Shell got down(10924110,41544343,true)
Shell got down(92041933,41544343,true)
true.
?-
```

Here the same parent process installs two monitors on the same child. When the child terminates after three seconds, two `down` messages are delivered, one per monitor. The distinct references (`Ref1` and `Ref2`) identify which monitor instance each message corresponds to.

Using the `monitor` option is the *safest* way to establish monitoring, because it is *atomic* with respect to process creation: either the child is spawned with monitoring already in place, or it is not spawned at all. In contrast, calling `monitor/2` after `spawn/2-3` is a separate step and therefore not atomic. This means that a very short-lived child may terminate in the gap between the `spawn` returning and the monitor being installed, in which case the parent will never receive a `down` message. For actors that may complete immediately (or fail immediately), `monitor(true)` avoids this race condition.

Monitoring can be stopped by passing the monitor ref to `demonitor/1-2`. When monitoring in installed using the `monitor` option, the pid of the monitored actor can also be used since it is identical to the ref.

The above example of the use of `monitor/2` was trivial, but in Chapter 5, both `monitor/2` and `demonitor/1-2` will be put to more serious use.

### Linking

If the value of the `link` option is `true` it means that when an actor terminates, any children that it might have are forced to terminate too. In our proposal, the default value for this option is `true`. So in the following example, when the outer call to `spawn/2` terminates, the nonterminating actor spawned by the inner call to `spawn/1` is automatically killed:

```
?- spawn(spawn(receive({})), Pid).
Pid = 10782012.
?-
```

Calling the built-in predicate `actors/1` confirms that only one actor is alive. We can infer that this must be the shell process, and that the actor created by the call `spawn(receive({}))` is gone:

```
?- actors(Alive).
Alive = [91033267].
?-
```

Linking in Web Prolog does *not* mean that a parent actor must terminate if any child that it may have spawned terminates:

```
?- spawn((spawn(exit(kill)),receive({})), Pid).
Pid = 40276924.
?-
```

Even though the child killed itself immediately, calling `actors/1` again shows that its parent `40276924` is still alive:

```
?- actors(Alive).
Alive = [91033267, 40276924].
?-
```

The link is *unidirectional,* and makes the life of a child dependent on the life of its parent, but never the other way around. (Erlang is different here, as its links are bidirectional. We will discuss this difference and justify our design choice in Section 2.5.)

The following example shows that when the `link` option is set to `false` the child (created by the inner call to `spawn/3`) does not necessarily terminate if the parent (created by the outer call to `spawn/1`) terminates:

```
?- spawn(spawn(receive({}),Pid,[link(false)])).
Pid = 10023812.
?- actors(L).
L = [91033267, 24917621].
?-
```

In our proposal, the default value for `link` is `true`, as this is usually what we want. In fact, and for a reason that will be explained in Chapter 4, in the context of distributed web programming it may often be a good idea to *require* that the value of `link` is set to `true`, and perhaps only allow the owner of a node to set it to `false`.

In later chapters we restrict who may disable lifetime coupling, especially for remote placement, since this interacts with resource control and deployment on the Prolog Web.

### Registering

Calling `register(Name,Pid)` associates the atom `Name` with `Pid`. The name can be used instead of the pid when calling `!/2`. For example, we can register our shell under the name `shell`:

```
?- self(Self), register(shell, Self).
Self = 85234512.
?- shell ! hello.
true.
?- spawn(shell ! goodbye).
true.
?- flush.
Shell got hello
Shell got goodbye
true.
?-
```

Registering is useful when we want to offer a service that should always be available under a name that is easy to remember. Should a crash occur, all our system needs to do is to restart it and associate the same name with the pid of the new process.

### Exiting

Web Prolog supports two predicates, `exit/1` and `exit/2`, that can be used to terminate an actor process.

If the process calls `exit(Reason)` it will terminate immediately, and if monitored by its parent process, the parent will be sent a `down` message, containing the term that `Reason` was bound to when predicate was called. Here is a simple and somewhat silly example where a process is spawned and monitored by the shell. Since the process is told to exit immediately with the reason `my_reason`, the shell receives a `down` message with the atom `my_reason` in its second argument:

```
?- spawn(exit(my_reason), Pid, [
       monitor(true)
   ]).
Pid = 91325643.
?- flush.
Shell got down(91325643,91325643, my_reason)
true.
?-
```

It sometimes happens that we need to terminate an actor process by force. Our echo server, for example, can only be terminated in this way. The predicate `exit/2` can be used to terminate any actor process with a known pid. Calling `exit(Pid,Reason)` sends an exit *signal* (not a message) to the actor with that pid, killing it.

If we do not know the pid, but it has a name that we know, we can use that instead. To see how this works, let us spawn a new monitored echo server and register it:

```
?- spawn(echo_actor, Pid, [
       monitor(true)
   ]),
```

```
   register(echo_actor, Pid).
Pid = 21562390.
?-
```

Now, let us say we change our minds and want to get rid of it again:

```
?- whereis(echo_actor, Pid),
   exit(Pid, 'We changed our minds!').
Pid = 21562390.
?- flush.
Shell got down(21562390,21562390,'We changed our minds!')
true.
?-
```

By calling `exit/2` with the pid in the first argument and a term detailing the reason for exiting in the second, a signal was sent to the process that terminated it. Note that a call to `exit/2` will only accept a pid in its first argument, so if all we have is a name, the built-in predicate `whereis/2` must be used to locate it.

Note that the reason passed to `exit/1-2` can be any term, not just an atom. This means that it can be used to transfer an arbitrarily large chunk of information back to the parent. However, in most cases an atom is all that is needed, and it is worth noticing that using `true` as a reason can be used to suggest to the parent that the termination was *normal*, even though it was caused by a call to `exit/1-2`.

What might seem a bit odd is that even thought `21562390` is now provably dead, trying to send it a message using a pid or calling `exit/2` still succeeds without an error, although no `down` message is being sent:

```
?- self(Self), $Pid ! echo(Self, bye).
Self = 85234512.
?- exit($Pid, 'We changed our minds!').
true.
?- flush.
true.
?-
```

Treating `!/2` and `exit/2` as no-ops when the pid points to a non-existent process is consistent with how it works in Erlang. However, trying to send a message using the *name* of a non-existent process generates an error:

```
?- echo_actor ! echo($Self, bye).
Error: The name 'echo_actor' is not associated with a pid.
?-
```

This too is consistent with how Erlang works.

**An hierarchy of actors**

When an actor process $A_1$ spawns an actor $A_2$, $A_2$ becomes the *child* of $A_1$, and $A_1$ the *parent* of $A_2$. Since $A_2$ may in turn spawn other processes, the actors involved may form a hierarchy.

```
demo :-
    spawn(parent).

parent :-
    spawn(child, Pid, [
        monitor(true)
    ]),
    receive({
        down(_, Pid, Why) ->
            format("~p ~p~n", [Pid, Why]) ;
        stop ->
            format("stopped~n")
    }).

child :-
    spawn(grandchild),
    receive({
        crash ->
            exit(crashed) ;
        stop ->
            format("stopped~n")
    }).

grandchild :-
    receive({stop -> true}).
```

In Erlang, things work a bit differently.

### 2.3.4 A closer look at receive/1-2

As shown already, but only for a trivial case, an actor process uses the receive primitive to extract messages from its mailbox. Since the syntax and semantics of `receive/1-2` is fairly complex, a closer look and more examples are needed. Below we give several simple examples illustrating different ways to use the `receive/1-2` predicate in Web Prolog, demonstrating how to handle different *types* of messages, use *timeouts*, apply *guards*, and more. Other examples illustrate how messages are deferred and handled in subsequent `receive/1-2` calls, demonstrating the flexibility of passing and processing messages in Web Prolog.

### Basic receive

In Web Prolog, just like in Erlang, the receive operation specifies an ordered sequence of *receive clauses* delimited by semicolons. A receive clause always has a *head* (a term) and a *body* of Prolog goals. Schematically, the basic form of a receive call looks like this:

```
receive({
    Head1 -> Body1 ;
    Head2 -> Body2 ;
    ...
    HeadN -> BodyN
})
```

Often, the head is just a single term serving as a *pattern*. Any term will do, ground or non-ground, and even a bare variable is fine.

As in Erlang, `receive/1` scans the mailbox looking for the first message (i.e. the oldest) that matches a pattern in any of the receive clauses, blocking if no such message is found. If a matching clause is found, the message is removed from the mailbox and the body of the clause is called. In Web Prolog, just like in Erlang, values of any variables bound by the matching of the pattern with a message are available in the body of the clause.

Often, a `receive/1` call waits for a specific message and executes the corresponding code in the body of the receive clause if a message that matches the pattern shows up. For example, the following call waits for a message in the form of `hello(Name)` and prints a greeting when it appears:

```
receive({
    hello(Name) ->
        format("Hello, ~s!~n", [Name])
})
```

We can specify multiple patterns in a single `receive/1` call. For example, this call handles messages of either the form `hello(Name)` or the form `goodbye(Name)`, but only one of them:

```
receive({
    hello(Name) ->
        format("Hello, ~s!~n", [Name]) ;
    goodbye(Name) ->
        format("Goodbye, ~s!~n", [Name])
})
```

We can use the `_` pattern to catch any message. In the following case, any message that does not match `hello(Name)` will be caught by the `_` pattern:

```
receive({
    hello(Name) ->
```

```
        format("Hello, ~s!~n", [Name]) ;
    _ ->
        format("Unknown message received.~n")
})
```

We can nest `receive/1-2` calls. Here, after having received the `start(Name)` message, the call waits for a `continue(Msg)` message:

```
receive({
    start(Name) ->
        format("Starting with ~s.~n", [Name]),
        receive({
            session(Msg) ->
                format("Continuing with ~s~n", [Msg])
        })
})
```

### Messages deferred

If no pattern matches a message in the mailbox, the message is *deferred*, which means that the message does not match any of the patterns in the current `receive/1-2` call and remains in the process's mailbox, possibly to be handled later in the control flow of the process. The receive is still running, waiting for more messages to arrive, and for one that will match. Some simple examples illustrating this behavior are shown below.

In the following example, the `goodbye("Bob")` message – which is the oldest and therefore first in line – does not match the clause in the first receive call and is deferred. The `hello("Alice")` message matches that clause, and then the clause in the second receive call handles the `goodbye("Bob")` message:

```
?- self(Self),
   Self ! goodbye("Bob"),
   Self ! hello("Alice"),
   receive({
       hello(Name1) ->
           format("Hello, ~s!~n", [Name1])
   }),
   receive({
       goodbye(Name2) ->
           format("Goodbye, ~s!~n", [Name2])
   }).
Hello, Alice!
Goodbye, Bob!
true.
?-
```

This behavior is particularly useful if we expect two messages but are not sure which one will arrive first. For example, if we insist on processing a message `foo` before `bar`, we can easily do that with receive, like so:

```
wait_foo :-                    wait_bar :-
    receive({                      receive({
        foo ->                         bar ->
            process_foo,                   process_bar
            wait_bar               }).
    }).
```

Even if `bar` arrives in the mailbox before `foo`, calling `wait_foo/0` would result in `foo` being selected and processed before `bar`. This is why the receive operator is often referred to as *selective* receive.

### Guards

The head of a receive clause can, in addition to the pattern, optionally specify a *guard* in the form of a Prolog goal. The role of a head of the form `Pattern if Goal` is to make pattern matching more expressive. Here is an example that distinguishes between positive and non-positive numbers using guards:

```
receive({
    number(N) if N > 0 ->
        format("Positive number: ~p~n", [N]) ;
    number(N) if N =< 0 ->
        format("Non-positive number: ~p~n", [N])
})
```

That was simple, and will work in Erlang too, but here is another example, using a different *kind* of goal after the `if` operator:

```
?- self(Self),
   Self ! hello(xantippa),
   receive({
       hello(W) if husband(W, H) ->
           format("Hello, ~w, say hello to ~w!~n", [W,H])
   }).
Hello, xantippa, say hello to socrates!
true.
?-
```

Note that the variable `H` does not occur in the pattern. A guard like that cannot be used in Erlang. In Web Prolog, its value can be passed to the body of the clause and do a job there. So while readers familiar with Erlang may wonder why we choose `if` instead of `when`, which is the operator Erlang is using, we just happen to think that this difference is significant enough to warrant a different name for the operator.

### Timeouts

The optional second argument of `receive/1-2` expects a list of options. The `timeout` option takes an integer or float that specifies the number of seconds before the call will succeed anyway, even if no match has been found. The `on_timeout` option takes a goal that is called if timeout occurs. Here is an example of its use:

```
receive({
    hello(Name) ->
        format("Hello, ~s!~n", [Name])
},[ timeout(5),
    on_timeout(format("No match received in 5 seconds.~n"))
])
```

If no message of the form `hello(Name)` is received within 5 seconds, the code in the `on_timeout` option is executed.

### An implementation of sleep/1

Here is how a predicate `sleep/1` that suspends execution `Time` seconds can be defined:

```
sleep(Time) :-
    receive({},[
        timeout(Time)
    ]).
```

The call to `receive/2` does not wait for messages, but when `Time` seconds have passed, it still succeeds.

### An implementation of flush/0

As we noted earlier, being able to inspect the contents of the shell's mailbox during interactive programming is important, and `flush/0` is a nice tool for doing just that. Its definition also serves as yet another example of the use of the `timeout` option:

```
flush :-
    receive({
        Message ->
            format("Shell got ~q~n",[Message]),
            flush
    },[
        timeout(0)
    ]).
```

The value `0` of the `timeout` option ensures that the loop terminates immediately if no messages remain in the mailbox. This is how the hanging of the `receive/1` call is avoided.

### A simple priority queue

To demonstrate the use of the `if` operator and the use of two `receive/2` options that causes a goal to run on timeout, we show a priority queue example borrowed from Fred Hébert's textbook on Erlang (Hebert, 2013). The purpose is to build a list of messages with those with a priority above 10 coming first:

```
important(Messages) :-
    receive({
        Priority-Message if Priority > 10 ->
            Messages = [Message|MoreMessages],
            important(MoreMessages)
    },[ timeout(0),
        on_timeout(normal(Messages))
    ]).

normal(Messages) :-
    receive({
        _-Message ->
            Messages = [Message|MoreMessages],
            normal(MoreMessages)
    },[ timeout(0),
        on_timeout(Messages=[])
    ]).
```

The timeout set to `0` means that it will occur immediately, but the system tries all messages currently in the mailbox first.

Below, we test this program by first sending four messages to the shell process, and then calling `important/1`:

```
?- self(S),
   S ! 15-high, S ! 7-low, S ! 1-low, S ! 17-high.
S = 34871244.
?- important(Messages).
Messages = [high,high,low,low].
?-
```

Note that the implementation of the priority-queue example relies on the deferring behavior of `receive/1-2` and would not work without it.

**The receive predicate is semi-deterministic**

The predicate `receive/1-2` is *semi-deterministic*, i.e. it either fails, or succeeds exactly once. The only way it will fail is if the goal in the *body* of one of its receive clauses fails, or if timeout occurs and the goal passed in the `on_timeout` option fails.

To see how it pans out in a simple corner case, consider the following two calls:

```
receive({foo(X) -> true})        receive({foo(X) -> fail})
```

The first call will succeed if a message matching the pattern `foo(X)` appears in the mailbox of the actor process executing the call, a term such as `foo(314)` for example. The second call will fail (and possibly cause backtracking) once `foo(314)` appears. Only by the first call will the variable `X` be bound (to `314`). Both calls will remove the matched message from the mailbox. In both cases, if a message appears that does not match the pattern, it is deferred.

To implement a looping behavior, Prolog programmers occasionally use a *failure-driven* loop that relies on backtracking rather than recursion. Using this technique, our echo server can be rewritten like so:

```
echo_actor  :-
    repeat,
    receive({
        echo(From, Msg) ->
            From ! echo(Msg),
            fail
    }).
```

For an Erlang programmer, this use of `receive/1` may come as a surprise and is not a technique that can be used in Erlang. In this particular case, a Web Prolog programmer would be advised to stick to the recursive version. However, there are cases when a failure-driven loop is the only way forward and we shall look at an important such case towards the end of this chapter.

## 2.4 Erlang-style programming examples in Web Prolog

> Reading the code was fun – I had to do a double take – was I reading Erlang or Prolog – they often look pretty much the same.
>
> *Joe Armstrong* (p.c. June 18, 2018)

Not only do Web Prolog programs *look* a lot like Erlang, they *behave* a lot like Erlang too. A good way to demonstrate this is to translate a fair number of different Erlang programming examples borrowed from tutorials and text books into Web Prolog and show that they run just like in Erlang. In this section we shall look at a count server, a fridge simulation, a universal stateful server with hot code swapping, actors playing ping-pong, and an approach to building simple supervision hierarchies. They are all

concurrent programs, but only locally so. (Distributed programming will be dealt with in Chapter 4.)

During the exploration of the examples, we point to the importance of the use of send and receive for implementing the *communication protocols* allowing actors acting as clients to, still within the bounds of one node, talk to actors acting as servers. We also explain why it is often a good idea to hide the details of such protocols from programmers behind a dedicated predicate API.

For more exhaustive documentation of all built-in predicates available in Web Prolog, consult Appendix A.

### 2.4.1 A count server

We have already looked at an example of a *stateless* server, namely the `echo_actor` presented earlier in this chapter. Let us now turn to *stateful* servers and show how state may be programmed. In the following example, we demonstrate how to script an actor that can keep a *count*:

```
count_actor(Count0) :-
    receive({
        count(From) ->
            Count is Count0 + 1,
            From ! count(Count),
            count_actor(Count) ;
        stop ->
            true
    }).
```

The predicate `count_actor/1` defines a loop where a call to the built-in predicate `receive/1` tries to select a message of the form `count(From)` from the mailbox, increment the counter, send the current count back to the actor referenced by the pid to which the variable `From` is bound, and continue looping by making a recursive call. Note how the state of the counter – the current count, that is – is kept in the argument of the predicate. Note also that we added a second receive clause that will allow us to terminate the server without using `exit/2`. We just have to send it a message of the form `stop`.

Here is how an actor following this script can be made to perform when spawning it:

```
?- spawn(count_actor(0), Pid, [
       monitor(true)
   ]).
Pid = 60367387.
?-
```

The `monitor` option was set to `true`, instructing the actor to send a special-purpose `down` message carrying a small piece of information on how the run went to its parent process just before terminating.

Calling `self/1` determines the identity of the toplevel process – the process that just became the parent of the spawned actor:[20]

```
?- self(Self).
Self = 41167597.
?-
```

In the next step the send operator `!/2` is used for sending a message to the spawned actor, instructing it to increment the count and to return the result in the form of a message:

```
?- $Pid ! count($Self).
true.
?-
```

A call to `receive/1` can be made in order to collect the message arriving from the actor:

```
?- receive({Count -> true}).
Count = count(1).
?-
```

Here, `!/2` is used to send two messages to the actor that will end up in its mailbox, in the order they were sent:

```
?- $Pid ! count($Self), $Pid ! stop.
true.
?-
```

Finally, the utility predicate `flush/0` is used to inspect the contents of the mailbox belonging to the toplevel process:

```
?- flush.
Shell got count(2)
Shell got down(60367387,60367387, true)
true.
?-
```

Because the actor was monitored, a `down` message was found in addition to the current count. The value `true` in the second argument means that `count_actor/1` succeeded.

[20] If you are an actor, this is how you find out who you are!

### 2.4.2 A bigger, tastier example

As a tastier example of how a process can be made to hold an updatable state during a conversation we have adapted a fridge simulation example from Fred Hébert's introduction to Erlang (Hebert, 2013):[21]

```
fridge(FoodList0) :-
    receive({
        store(From, Food) ->
            self(Self),
            From ! Self-ok,
            fridge([Food|FoodList0]);
        take(From, Food) ->
            self(Self),
            (   select(Food, FoodList0, FoodList)
            ->  From ! Self-ok(Food),
                fridge(FoodList)
            ;   From ! Self-not_found,
                fridge(FoodList0)
            );
        terminate ->
            true
    }).
```

The program creates a process allowing three operations: storing food in the fridge, taking food from the fridge, and terminating the fridge. It is only possible to take food that has been stored beforehand. Again, with the help of recursion the state of a process can be held entirely in the argument of the predicate. In this case we choose to store all the food as a list, and then look in that list when someone needs something.

Assuming the above program is already available, the following session creates the server process. We can then call `self/1`, `!/2` and `flush/0` from the shell in order to simulate the actions of a client:

```
?- spawn(fridge([]), Pid, [
       monitor(true)
   ]).
Pid = 77346122.
?- self(Me), $Pid ! store(Me, meat), $Pid ! store(Me, cheese).
Me = 97216744.
?- flush.
Shell got 77346122-ok
Shell got 77346122-ok
true.
```

[21] `http://learnyousomeerlang.com/more-on-multiprocessing#state-your-state`

```
?- self(Me), $Pid ! take(Me, cheese).
Me = 97216744.
?- flush.
Shell got 77346122-ok(cheese)
true.
?- $Pid ! terminate.
true.
?- flush.
Shell got down(77346122,77346122,true)
true.
?-
```

### 2.4.3 Hiding the details of protocols

In the previous example, we expected programmers to know the details of the protocol that must be followed when interacting with our fridge simulation, which forced them to make raw calls using the send operator in combination with the `flush/0` utility predicate. As suggested by Fred Hébert in his book, that is often a useless burden, and good way around it is to abstract messages away with the help of predicates (or in Hébert's case, functions) dealing with receiving and sending them:

```
store(Pid, Food, Response) :-
    self(Self),
    Pid ! store(Self, Food),
    receive({
        Pid-Response -> true
    }).

take(Pid, Food, Response) :-
    self(Self),
    Pid ! take(Self, Food),
    receive({
        Pid-Response -> true
    }).
```

Calling `receive/1` immediately after having sent the message guarantees that the communication between client and server stays synchronous. Using `store/3` and `take/3`, the interaction with the fridge becomes somewhat easier:

```
?- spawn(fridge([]), Pid, [
       monitor(true)
   ]).
Pid = 55289322.
?- store($Pid, cheese, Response).
```

```
Response = ok.
?- take($Pid, cheese, Response).
Response = ok(cheese).
?-
```

When dealing with actors with more complex protocols, such abstractions turns out to be of considerable value.

### 2.4.4 Using delayed sending

Appendix A documents a built-in predicate `send/2-3` that allows a process to delay the sending of a message, but also to cancel the sending if needed, by calling `cancel/1`. This can be used to implement an alarm that can be canceled, like so:

```
alarm :-
    receive({
        ring ->
            writeln('Alarm ringing!'),
            alarm;
        stop ->
            true
    }).
```

Here we schedule the alarm to ring in 5 seconds, with an `id` set to `alarm1`. Then we cancel the alarm after a couple of seconds.

```
?- spawn(alarm, Pid).
Pid = 60034123.
?- send($Pid, ring, [
       delay(5),
       id(alarm1)
   ]).
?- cancel(alarm1).
true.
?-
```

If the cancelation succeeds, the message is withdrawn and the alarm never rings. If `cancel/1` is called too late, the alarm message has already been delivered and "Alarm ringing!" will be printed.

### 2.4.5 Prolog actors playing ping-pong

Since the servers in the previous sections are running in parallel to the shell and are talking to it using asynchronous messaging, we have already demonstrated the use

of concurrency. Below, in a probably more convincing example inspired by a user's guide to Erlang,[22] two processes are first created and then start sending messages to each other a specified number of times:

```
ping(0, Pong_Pid) :-
    Pong_Pid ! finished,
    format('Ping finished.~n',[]).
ping(N, Pong_Pid) :-
    self(Self),
    Pong_Pid ! ping(Self),
    receive({
        pong ->
            format('Ping received pong.~n',[])
    }),
    N1 is N - 1,
    ping(N1, Pong_Pid).

pong :-
    receive({
        finished ->
            format('Pong finished.~n',[]);
        ping(Ping_Pid) ->
            format('Pong received ping.~n',[]),
            Ping_Pid ! pong,
            pong
    }).

ping_pong :-
    spawn(pong, Pong_Pid),
    spawn(ping(3, Pong_Pid)).
```

When `ping_pong/0` is called the behavior of this program exactly mirrors the behavior of the original version in Erlang:

```
?- ping_pong.
true.
Pong received ping.
Ping received pong.
Pong received ping.
Ping received pong.
Pong received ping.
Ping received pong.
Ping finished.
Pong finished.
?-
```

[22] See `http://erlang.org/doc/getting_started/conc_prog.html`

This is a concurrent program, but its execution is not done in parallel, but rather as is illustrated in Figure 2.3.

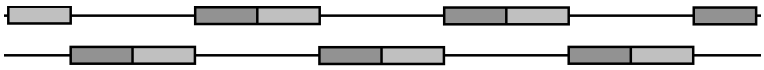


**Fig. 2.3** A timeline where the upper line represent the "pinger" and the lower line the "ponger," and where each gray bar represent a task consisting of the sending of a message (in light gray) and the reception of the response (in darker gray)

In Appendix D a slightly modified version of this program is used for benchmarking send and receive in Web Prolog. It turns out that if we run this on a 2023 Apple iMac with the M3 chip with `N` set to `100,000` instead of `3` it runs in less than one second. This means that each task is performed in less than 5 microseconds. Appendix D also shows that Erlang is even faster.

### 2.4.6 Event-driven state machines

Building on the guard mechanism introduced in Section 2.3.4, we now show a simple but representative application: an event-driven state machine whose transitions can consult the process's knowledge base. The key point is that clause selection can depend not only on the incoming message pattern, but also on arbitrary relational conditions, and any bindings produced by those conditions can be used by the transition action.

Web Prolog's `receive` primitive can implement such machines directly. Message patterns represent events, guards represent transition conditions, and clause bodies implement actions and state updates. Figure 2.4 depicts a machine with two states and four transitions.

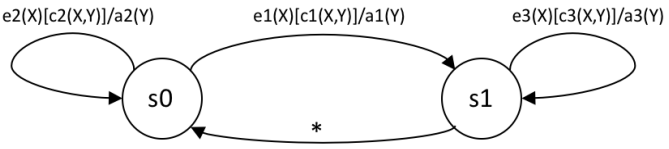



**Fig. 2.4** A simple event-driven state machine.

Using predicate names to encode states, one possible implementation is:

```
s0 :-
    receive({
       e1(X) if c1(X,Y) ->
```

```
s1 :-
    receive({
       e3(X) if c3(X,Y) ->
```

```
        a1(Y),                          a3(Y),
        s1 ;                            s1 ;
    e2(X) if c2(X,Y) ->             _AnyEvent  ->
        a2(Y),                          s0
        s0                      }).
}).
```

In state `s0` the machine waits for an event message of the form `e1(X)` or `e2(X)` in the current process's mailbox. If `e1(X)` matches and `c1(X,Y)` succeeds, the action `a1(Y)` is executed and the machine transitions to `s1`. If `e2(X)` matches and `c2(X,Y)` succeeds, the action `a2(Y)` is executed and the machine remains in `s0`. The other state is analogous: `e3(X)` triggers a guarded self-loop in `s1`, while any other event falls back to `s0`.

Seen from an Erlang perspective, the example also clarifies the design choice: Erlang restricts guards to a small, carefully controlled language, whereas Web Prolog permits relational guards that consult application predicates and introduce new bindings, as in the `husband/2` example earlier in this chapter. This expressiveness is exactly what makes knowledge-base-dependent transitions feel natural, but it also places responsibility on the programmer to keep guards efficient and free of side effects.

It is worth noting that event-driven state machines have historically been less prominent in the Prolog world. One likely reason is that message passing arrived relatively late and unevenly across Prolog systems, gaining wider traction only with thread and message-passing facilities such as those explored in the ISO Prolog Threads draft.[23] By contrast, Erlang/OTP treats this programming model as foundational, as exemplified by `gen_statem`.[24] In Chapter 6 we build on this observation and propose Web Prolog as a scripting language for StateChart XML (SCXML), a W3C standard for event-driven state machines of a more sophisticated kind.

===

Erlang enforces purity and efficiency by only allowing a restricted set of primitives in guards, and completely disallows calling user-defined functions. For the use cases the people behind Erlang had, it probably made sense to impose such restrictions.

In Web Prolog, however, although only the first solution will be searched for, *any* goal may be used as a guard and values of variables bound by it are available in the body. This makes the receive construct more powerful than in Erlang, but it also means that the programmer is made responsible for keeping guards as simple and efficient as possible and to avoid side effects. In our opinion, and as exemplified by the state-machine code in Section 2.4.6, enabling the programmer to condition the matching of a receive clause on the content of the entire Prolog database makes it worth it.

Most Erlang developers accept that guards serve a specific role, and they understand the trade-offs. But some find it limiting and express desire for more expressive,

---

[23] `http://logtalk.org/plstd/threads.pdf`

[24] `http://erlang.org/doc/man/gen_statem.html`

reusable guard logic. So there is broad interest and research into relaxing the restriction for *pure* functions.

—

The change of operator names from `when` to `if` suggests the change of semantics. (This is not the only reason why we prefer `if`.)

### 2.4.7 Getting answers through backtracking

We now turn to another feature that Web Prolog has but that Erlang lacks, and give an example that rely on semi-deterministic message reception – a receive either fails, or succeeds exactly once – to structure clean, failure-driven loops that interact predictably with asynchronous messaging, including stepping through solution streams and a small toplevel-style interaction.

As stated previously, at least in theory, unexpected interactions between language features and possible impedance mismatches between Prolog's relational, nondeterministic programming model and Erlang's functional and message passing model should not cause any problems. As we now turn to more examples that go beyond what Erlang can easily do, we need to see how well the Erlang-style constructs do mix with backtracking for example? In this section we show some examples suggesting that the mix is both sound and easy to understand.

Suppose the query given in the argument to `spawn/2` has several answers, a query such as `?-human(Who)` for example. Below, a goal containing this query is called, the first solution is sent back to the calling process, and `receive/1` is then used in order to listen for a message of the form `next` or `stop` before terminating:

```
?- self(Self),
   spawn(( human(Who),
           Self ! Who,
           receive({
               next -> fail ;
               stop -> true
           })
         ), Pid).
Pid = 76123351,
Self = 90054377.
?- flush.
Shell got socrates
true.
?- $Pid ! next.
true.
?- flush.
Shell got plato
true.
?- $Pid ! stop.
```

```
true.
?-
```

As this session illustrates, the spawned goal generated the solution `socrates`, sent it to the mailbox of the parent shell process, and then suspended and waited for more messages. When the message `next` arrived, the forced failure triggered backtracking which generated and sent `plato` to the mailbox of the toplevel shell process. The next message was `stop`, so the spawned process terminated. Note that the example demonstrated an actor adhering to what might be seen as a tiny communication protocol accepting only the messages `next` and `stop`.

Looking at the code in the first argument of the calls to `spawn/2` above, this is how we in Prolog often loop over the solutions to a query, using a failure-driven loop rather than a recursive one. Again, the most obvious way to make the code work as expected, is to allow receive to fail.

One needs to observe, however, that the goal to be solved in the above example is hard-coded into the program, that the protocol for the communication between client and actor is overly simplistic, and that neither failure of the spawned goal, nor error thrown by it, are handled. There is clearly a need for something more complete and more generic.

### 2.4.8 The toplevel behavior: a preliminary sketch

In essence, a Prolog toplevel is a failure-driven loop running inside another loop – recursive or failure driven. The failure-driven inner loop allows a client to ask for one solution at a time to a goal, while the outer loop allows it to call several goals in a row and run each of them to completion.

Below, we show how we can build a simple Prolog toplevel by using a meta-predicate (such as `call_cleanup/2`) and by specifying a small set of custom messages carrying answers and/or the state of the process that needs to be returned to the calling process. The predicate `call_cleanup/2` is here used not only to call a goal, but also to check if any choice points remain after the goal has been called or backtracked into.[25]

```
toplevel(Pid) :-
    toplevel(Pid, []).

toplevel(Pid, Options) :-
    self(Self),
    spawn(session(Pid, Self), Pid, Options).

session(Pid, Parent) :-
    receive({
```

[25] `http://www.swi-prolog.org/pldoc/man?predicate=call_cleanup/2`

```
        '$call'(Template, Goal) ->
            (   call_cleanup(Goal, Det=true),
                (   var(Det)
                ->  Parent ! success(Pid, Template, true),
                    receive({
                        '$next' -> fail ;
                        '$stop' -> true
                    }),
                    !
                ;   Parent ! success(Pid, Template, false)
                )
            ;   Parent ! failure(Pid)
            )
    }),
    session(Pid, Parent).
```

A suitable predicate API hiding the details of the protocol from the programmer can be written like so:

```
toplevel_call(Pid, Template, Goal) :-
    Pid ! '$call'(Template, Goal).

toplevel_next(Pid) :-
    Pid ! '$next'.

toplevel_stop(Pid) :-
    Pid ! '$stop'.
```

Here is a session that shows that these predicates work as intended:

```
?- toplevel(Pid, [
       monitor(true)
   ]).
Pid = 23891373.
?- toplevel_call($Pid, Who, human(Who)).
true.
?- flush.
Shell got success(23891373,socrates,true)
true.
?- toplevel_next($Pid).
true.
?- flush.
Shell got success(23891373,plato,true)
true.
?- toplevel_next($Pid).
true.
?- flush.
```

```
Shell got success(23891373,aristotle,false)
true.
?-
```

Note that because of the outer recursive loop, our simple toplevel actor is again poised to run another query:

```
?- toplevel_call($Pid, X, q(X)).
true.
?- flush.
Shell got down(23891373,23891373,existence_error(proc,q/1))
true.
?-
```

Here the actor crashed, and the reason is that the code for `toplevel/1-2` does not say anything about what should happen if an error is thrown, which is the case if the predicate called by the goal is not defined. Since the spawned process is monitored, however, the error message did eventually reach the mailbox of the spawning process anyway, in the form of a `down` message.

A similar problem is evident should we tell the actor to execute a query that does not terminate. Using `exit/2` works, but will once again kill the entire actor process.

Such problems can be fixed. As it turns out, however, and as will be demonstrated in Chapter 3, Web Prolog programmers do not need to define their own toplevel predicates. Instead, they can use a set of built-in predicates in order to create and control more powerful Prolog toplevels, which are actors adhering to a much improved protocol. In the terminology introduced above, these are Erlang-style behaviors implementing a standardized toplevel protocol.

## 2.5 Summary

This chapter introduced Web Prolog as a hybrid language that extends a substantial subset of ISO Prolog with Erlang-style constructs for programming concurrency. The core additions – actors with private state, asynchronous message passing (`!/2`), process creation (`spawn/2-3`), and fault-handling primitives such as links, monitors, and exit signals. A program that never mentions these constructs behaves like a conventional Prolog program, while programs that do use them gain access to an Erlang-like world of concurrent processes.

We also argued that Erlang provides an unusually suitable point of departure for such an extension. Its actor model is mature, battle-tested, and close enough in syntax and spirit to Prolog that both languages can share patterns, idioms, and even didactic material. Many of the examples in this chapter are thin Prolog translations of familiar Erlang patterns. The combination of Prolog's backtracking search with Erlang-style concurrency is made workable by allowing receive to be semi-deterministic and by separating the nondeterminism of search from the determinism of message

handling. This makes it possible to treat a Prolog toplevel as just another actor and to encapsulate interactive sessions as behaviors implemented on top of a common concurrency substrate.

Taken together, these choices define Web Prolog as a multi-paradigm language. Its logical expressivity remains that of ISO Prolog, but its operational repertoire is enlarged by an actor-based concurrency model that is both network-transparent and compatible with the rest of the Prolog Trinity. There are other possible ways of adding concurrency to Prolog, but the thesis of this chapter is simply that Erlang-style actors provide a coherent and practically manageable foundation on which the subsequent chapters can build.

# Chapter 3
# Prolog agents

> Imagine a world of **Prolog agents**, some useful, others playful, bringing joy to games and virtual worlds; some short-lived, others long-lived, some simple, others complex – but maybe built from simpler ones. Written in Web Prolog, talking Web Prolog with other agents, using Web Prolog knowledge bases to guide their actions and conversations, making sure important capabilities of clever conversational agents, such as natural language understanding, knowledge representation, reasoning and real-time interaction, are accounted for.
>
> *Prolog agents – the elevator pitch*

Chapter 1 introduced the notion of a Prolog agent, and Chapter 2 the more precise concept of a Prolog actor – the most elementary form of Prolog agent in the Prolog Trinity ecosystem. In this chapter, the concept of a Prolog agent is further developed and two important kinds of agents, Prolog toplevels and Prolog nodes, will be introduced and their roles in the ecosystem described.

The reason for referring to both actors and nodes as agents is to focus on their similarities. The similarity that makes us refer to both of them as Prolog agents is that they are both processes that are capable of talking Prolog to other agents. Also, they both live on the Prolog Web. There are of course differences as well. A node is typically long-lived. It is capable of serving many clients at the same time. It can be programmed, but only by its owner.

## 3.1 Prolog agents in a nutshell

In this book, an *agent* is understood in a deliberately modest sense: a persistent software process that participates in ongoing interaction with an external environment.[1]

[1] In this spirit we follow Russell and Norvig (2016) in treating "agent" as an analytical lens rather than an absolute classification.

**Prolog agents.** A *Prolog agent* is a process on the Prolog Web whose interactions are expressed as Prolog terms: it receives goals or messages and responds with answers, messages, or effects in the same term-based form. Being a Prolog agent is therefore an interface and behaviour property, not a requirement about implementation language.

**State and interaction.** Agents are persistent and interactive, but they need not be stateful. Some are effectively stateless, while others maintain state in receive-loop parameters, control structure, or a private dynamic database. The private database is thus a capability, not a defining feature.

**Actors and nodes.** We treat Prolog actors and Prolog nodes uniformly as agents because they are all addressable, persistent, and interactive, and because their interactions are mediated by Prolog terms. They differ in role: actors are loci of concurrent computation; toplevels expose a conversational interface; nodes host agents and knowledge, expose web APIs, and enforce policy.

**Private beliefs and shared knowledge.** Each agent may combine a private workspace (including an optional private dynamic database) with shared node-provided knowledge (typically a read-only database maintained by the node owner). Private definitions can shadow shared ones, allowing local specialisation without modifying the shared base.

**Protocols as contracts.** Toplevel actors are special because their behaviour is governed by an explicit, observable protocol. The Prolog Toplevel Communication Protocol (PTCP) specifies how goals are submitted, how solutions are delivered incrementally, and how exceptional situations such as aborts and exits are handled. This turns REPL interaction into a first-class behavioural contract and supports later discussions of timeouts and of stateless versus session-oriented interfaces.

The following sections refine this picture by examining actor agents and their life-cycle, then toplevel agents and PTCP, and finally nodes as agents in the Prolog Web.

## 3.2 More about Prolog actor agents

Actor agents are the workhorses of the concurrent strand of the Trinity: they execute Prolog code concurrently, interact by exchanging Prolog terms, and may maintain actor-local state across repeated interactions. This section makes that operational picture more concrete. We first introduce the actor's private Prolog database and the `load_*` options used to initialise it. We then sketch the life-cycle of a typical actor and explain how receive loops realise explicit communication protocols. Finally, we discuss the dynamic (still private) database as a non-logical but sometimes convenient mechanism for actor-local updates.

### 3.2.1 An actor agent is equipped with a private Prolog database

When running the examples in Chapter 2 we were somewhat vague about the origin of the code being executed. The only clue provided was that it might reside in the node's shared database, which defines read-only data and programs accessible to any actor living on that node. While we will delve deeper into the shared database later in this chapter, we first turn our attention to the other storage area – the one located *within* the actor itself and where programs and data exclusive to that actor are stored. While all actors have a private database, it starts out empty by default.

During the spawn operation, the (soon-to-be) parent process can initialize the private database of the (soon-to-be) child actor with clauses defining one or more predicates. Any predicate defined in the private database will take precedence over and shadow corresponding predicates in the shared database, ensuring the child actor can operate with its own customized logic and data.

Consider the following example where the `load_text` option is passed to `spawn/3`, specifying that the source code for our count server should be loaded into the actor's private database before calling the goal in the first argument:

```
?- spawn(count_actor(0), Pid, [
       load_text("
           count_actor(Count0) :-
               receive({
                   count(From) ->
                       Count is Count0 + 1,
                       From ! count(Count),
                       count_actor(Count) ;
                   stop ->
                       true
               }).
       "),
       monitor(true)
   ]).
Pid = 45092311.
?-
```

In addition to the `load_text` option, three other options can be used to populate the private database of an actor. The `load_list` option loads a list of clauses or directives, the `load_uri` option loads the content specified by a URI, and `load_predicates` takes a list of predicate indicators and loads the clauses for the indicated predicates that are accessible by the caller. See Section A.1 for more information.

The clauses in an actor's private database represent its unique beliefs and skills, distinguishing them from the shared beliefs and skills accessible to other actor agents on the same node. In this sense, the private database can be seen as the core of the actor's individuality, encapsulating the knowledge and capabilities that are exclusive to it.

The `load_*` options make a kind of sharing of beliefs and skills between a parent process and a child process possible. The child is "born" with these beliefs and skills, they are "innate." It is up to the parent actor to determine exactly what should be passed on to the child actor.

At this point, and practically speaking, the actor's private database and the `load_*` options may not seem all that useful. After all, it seems we could just as well have stored the implementation of `count_actor/1` in the node's shared database. As we shall see, they are more useful in the distributed case, for shipping predicates from a client to a node, or from a node to a client, and execute them there.

### 3.2.2 The life-cycle of a typical Prolog actor agent

> We have the black boxes that do things, and we have the messages in between. What goes on inside the black boxes does not really matter; what takes place there is abstracted over.
>
> *Joe Armstrong*

The diagram in Figure 3.1 illustrates the life-cycle of a typical Prolog actor.

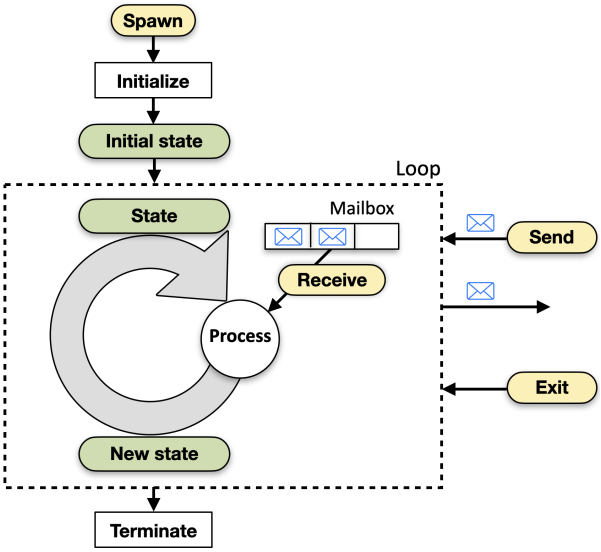



**Fig. 3.1** Lifecycle of a Prolog actor: spawn, interact, terminate, and clean up

When an actor is spawned, it first passes through an initialization phase where it is configured in a way that will suit its intended purpose. This includes specifying values for options such as `monitor`, `link` as well as the `load_*` options.

For example, when the count actor in the previous sections was spawned, the source code specified by the `load_text` option was loaded into the actor's private Prolog database and the `monitor` options instructed the actor to send a `down` message to the shell just before terminating. The call to `spawn/3` succeeded, binding the variable in the second argument to a pid. By this, the initialization phase is completed, the actor is now in its *initial state*, and the loop that implements the actor's *communication protocol* is entered. Typically, processing stops at a call to `receive/1-2` where it blocks, waiting for a `count` message or a `stop` message to appear in its mailbox.

The notion of a communication protocol is central to Erlang-style concurrent programming. The loop implements the communication protocol that clients that want to interact with the actor must follow. The interaction is driven by coordinating send and receive operations, with the protocol defining the rules. The sending process transmits messages in the specified format, and the receiving process captures and interprets them. They must adhere to the syntax, semantics and timing constraints of the communication protocol:

- **Syntax**: This refers to the structure and format of the data exchanged between processes. It dictates how data is packaged, including the organization of various data elements.
- **Semantics**: This involves the meaning and interpretation of the exchanged data. It defines the rules for understanding the content of the message, what actions should be triggered upon receiving certain data, and how to handle errors or unexpected data.
- **Timing**: This concerns the timing aspects of the communication, such as when data is sent, the pace of data exchange, and managing the order and sequence of messages so that both processes can coordinate despite delay, reordering, and failure.

The Web Prolog receive operator, while blocking the process at the point of a `receive/1-2` call, allows for the implementation of very flexible communication protocols through mechanisms like selective pattern matching and timeouts. The selective pattern matching allows the process to wait for a particular message but can also defer out-of-order messages for later processing. By specifying a timeout to avoid the receive call being blocked indefinitely, alternative actions can be taken if an expected message does not arrive in time. This helps reduce the chance of breakdowns and allows for more adaptive communication.

Whenever the actor runs a `receive/1-2` call, a message is selected from the mailbox and processed sequentially. The result often depends on the state of the process, and as part of processing, a new state for the actor may be computed. This will be the current state for processing the next message selected.

Some actors are stateless (e.g. the echo actor), and those that are stateful represent their state and perform state transitions in a variety of ways: as values threaded through the arguments of a receive loop (for example an integer updated by calls to `is/2` in the count actor, or a list of atoms in the fridge simulation), as the explicit states and transitions of a state machine, or implicitly through the choice points

created by a failure-driven loop. These techniques all encode state in the control flow or data flow of the program itself. In addition to these, Web Prolog offers a more direct – though non-logical – mechanism for maintaining actor-local state: a private dynamic Prolog database.

### 3.2.3 The dynamic (and still private) Prolog database

As long as the actor process is alive, it is allowed to update its private database using predicates such as `assert/1`, `retract/1` and `retractall/1`. Following the ISO standard, predicates that are modified this way need to be declared using the `dynamic/1` directive. Updates only affect that actor's private database: they are not visible to other actors running on the same node.

Because the dynamic database is strictly actor-local, it neither introduces shared-state races nor provides a mechanism for shared-state inter-process communication.

For example, although the following goal is permitted, it is of limited use: the clause `foo(a)` exists only inside the spawned actor and disappears when that actor terminates.

```
?- spawn(assert(foo(a)), Pid).
Pid = 32861299.
?-
```

The dynamic database provides another way to transfer from one state to the next. This is often a bad idea.

Throughout the Prolog Trinity ecosystem, private state is treated as volatile and local to an agent, whereas shared data is governed by explicit node-level policies and exposed under controlled conditions.

#### More about the dynamic database

The dynamic database is a non-logical extension to Prolog, and its use for managing state is often viewed with suspicion by Prolog programmers. The reason is well known: changes to the database are not reverted on backtracking, which makes dynamic predicates behave like mutable global variables and exposes programs to the usual problems associated with such entities.

At first glance, this raises the question of whether the dynamic database could lead to problems in Web Prolog, especially in a concurrent setting. The short answer is no. Although Erlang does not offer a Prolog-style dynamic database, it provides a closely related mechanism in the form of the *process dictionary*. Each Erlang process maintains a private key–value store that cannot be accessed or modified by other processes. As with the dynamic database, its use is generally discouraged, since it undermines referential transparency and makes programs harder to reason about and debug.

The similarity is hard to miss. This suggests that the dynamic database in a Web Prolog actor is not inherently more dangerous than the process dictionary in Erlang. At the same time, the dynamic database is arguably more expressive and convenient, which also means that programmers may be tempted to rely on it more often than is advisable.

While both the dynamic database in Web Prolog actors and the Erlang process dictionary are strictly private, Erlang also provides mechanisms for mutable *shared* state. These include Erlang Term Storage (ETS), which offers in-memory tables with configurable access rights, and Mnesia, a distributed database supporting transactions, persistence, and replication. More recently, Erlang has also introduced `persistent_term`, which provides efficient read-only access to immutable global data, at the cost of expensive updates.

This raises a broader design question for Web Prolog: should it offer forms of dynamic storage beyond what can be represented by predicate arguments or asserted in an actor's private database? Given Web Prolog's focus on the Web, a transactional RDF store would be a natural candidate, and such a solution is readily available in systems like SWI-Prolog. A Linda-style blackboard is another possible direction. For the time being, however, Web Prolog deliberately restricts dynamic state to the private scope of individual agents, leaving shared mutable state to be addressed by explicit, higher-level mechanisms.

## 3.3 Prolog shells and other toplevel actors

As we saw already in Chapter 2, calling `self/1` in a terminal binds its argument to the pid of the currently running shell process:

```
?- self(Pid).
Pid = 72097632.
?-
```

Here, the actor with the pid `72097632` is a shell. Shells are toplevels, and therefore also agents. Shells handle all the things that toplevels handle, but also add a number of useful things for when we are talking to Prolog over a terminal. This includes I/O (read and write) and utilities such as `flush/0` and the dollar notation. Such features are supplied by the *node controller* rather than a toplevel alone. (We will say more about the node controller further ahead in this chapter.)

Here is how we instruct the shell to spawn a new toplevel:

```
?- toplevel_spawn(Pid).
Pid = 16226587.
?-
```

Note that `16226587` is a toplevel actor but not a shell. As with every other Prolog actor, it comes with a mailbox, a private Prolog database, the ability to send messages to other actors, as well as the ability to create other actors.

The feature that distinguishes a toplevel from other actors is that it comes with a standardized built-in special-purpose communication protocol.

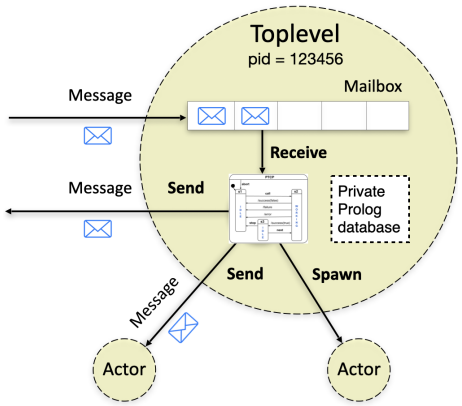



**Fig. 3.2** The anatomy of a Prolog toplevel actor.

### 3.3.1 A Prolog toplevel is an actor with a built-in protocol

A programmer firing up a traditional Prolog system is likely met with a query prompt. In the literature, this is usually referred to as the *toplevel*. The reason we refer to it a *shell* is because we want to use the term *toplevel* to refer to toplevels that are not shells.

In traditional Prolog, a program cannot *internally* create a toplevel, pose queries and request solutions on demand, but this is something that Web Prolog allows. In traditional Prolog the shell is *lazy* in the sense that new solutions to a query are only computed on demand. As we shall see, this is how the Web Prolog toplevel actor works too.

In Web Prolog, a toplevel actor is a programming abstraction modelled on the interactive toplevel of Prolog. A toplevel actor is like a first-class interactive Prolog toplevel, accessible from Web Prolog as well as from other programming languages such as JavaScript. We can also think of it as an *encapsulated Prolog session*, an abstraction aiming at making Prolog programmers feel right at home.

A toplevel is a kind of actor, and what distinguishes it from other kinds of actors is the *protocol* it follows when it communicates, i.e. the kind of messages it listens for, the kind of messages it sends and in what order, and the behavior this gives rise to.

The protocol must not only allow a client to submit queries and a toplevel to respond with answers, it must also allow the toplevel to prompt for input or produce

output at any time, in an order and with a content as dictated by the program that it runs. All toplevels follow this protocol. The terminal adheres to it as well, and even a human user of a terminal talking to a shell must adapt to it in order to have a successful interaction.

The design of the toplevel actor in Web Prolog is in fact very much inspired by the informal communication protocol that we as programmers adhere to when we invoke a Prolog shell from our OS prompt, load a program, submit a query, are presented with a solution (or a failure or an error), type a semicolon in order to ask for more solutions, or hit return to stop. These are "conversational moves" that Prolog understands. There are even more such moves, because after having run one query to completion, the programmer can choose to submit another one, and so on. The session does not end until the programmer decides to terminate it. There are only a few moves a client can successfully make when the protocol is in a particular state, and the possibilities can easily be described, by a state machine for example, as will be shown in the next section.

### 3.3.2 The Prolog Toplevel Communication Protocol

The diagram in Figure 3.3 is a statechart specifying the Prolog Toplevel Communication Protocol (PTCP) – a protocol for the communication between a client and a toplevel actor. The client can be any process (including another actor or (say) a JavaScript process) capable of sending the messages and signals in bold to the toplevel. The toplevel actor is responsible for returning the messages with a leading / back to the client. The use of a statechart allows us to show that no matter the current state of the protocol, **abort** will always take it to the state from which a new goal can be called and **exit** will always terminate the toplevel process.

A process is able to *spawn* a toplevel. After having done so, the process becomes the parent of the toplevel and can start communicating with it according to the protocol. Initially, the protocol is in state **s1**, where the toplevel rests idle, waiting for a **call** message containing a goal. When such a message arrives, the protocol transitions to state **s2**, where the toplevel is doing actual work. The protocol will remain in state **s2** until some work is done and the toplevel sends a message indicating either `/success`, `/failure`, `/error`, or (if the process is monitored) `/down` to the client. On the event of the toplevel sending `/failure` or `/error`, the protocol will transition back to state **s1**. This will be the case also for a `/success` message that indicates no more solutions to the query can be found (marked with false in the chart). However, if a `/success` message indicates more solutions may exist (marked with true in the chart), the protocol transitions to state **s3**, where it will wait for a message **next**, **stop**, **abort** or **exit** to arrive from the client. If the message is **stop** or **abort** the protocol will transition back to state **s1**, if it is **next** it will transition to state **s2** and trigger the search for more solutions, and finally, if it is **exit**, it will (if the process is monitored) force the toplevel to send a message `/down` back to the

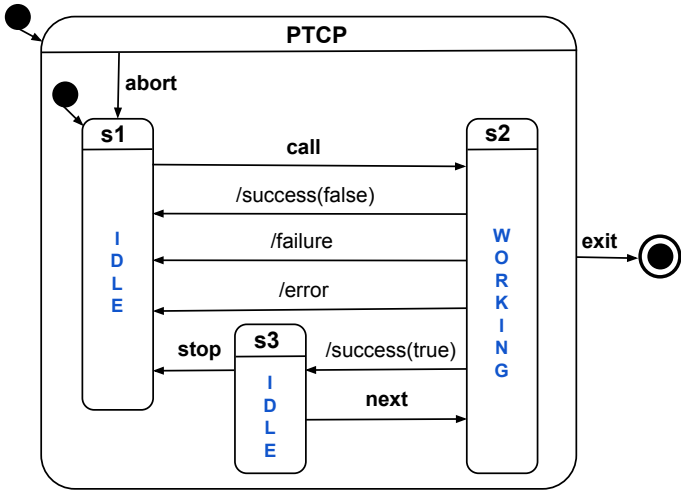



**Fig. 3.3** Statechart for the Prolog Toplevel Communication Protocol (PTCP): client moves and toplevel responses.

client and then to terminate. Any other messages will be deferred to the code being executed by the toplevel.

Web Prolog comes with built-in predicates which allow a client to spawn a toplevel actor and interact with it during its life-cycle:

```
toplevel_spawn/1-2    toplevel_call/2-3    toplevel_next/1-2
toplevel_stop/1       toplevel_abort/1
```

As witnessed by some of the examples in Chapter 2, such predicates can be defined in terms of the more primitive `spawn/1-3`, `!/2` and `receive/1-2` predicates. The predicates proposed here come with a number of important additional options, the use of which will be demonstrated in the sections that follow. Appendix A contains an excerpt from the (draft) manual which covers most of the details of our proposal.

### Toplevel actors are stateful

Clearly, a Prolog toplevel is a kind of server (in the sense of Erlang – see Section 2.4.1). Also, note that a toplevel, even when not explicitly threading any state, nor using the dynamic database, must still be considered stateful. Rather than using an explicit data structure for holding the state, it is the underlying Prolog process *as such* that holds it, most clearly shown in the way the toplevel "remembers" its history and how its behavior is influenced by this, enabling it to react appropriately when a client requests the next solution to a query.

### 3.3.3 Signatures and options for `toplevel_*` predicates

**`toplevel_spawn(-Pid, +Options?)`**

| Option | Values | Default |
|---|---|---|
| `session` | `true`, `false` | `false` |
| `target` | Pid | caller's pid |

**`toplevel_call(+Pid, +Goal, +Options?)`**

| Option | Values | Default |
|---|---|---|
| `template` | term | Goal |
| `offset` | integer $\geq 0$ | `0` |
| `limit` | integer $> 0$ | all |
| `once` | `true`, `false` | `false` |
| `target` | Pid | from `toplevel_spawn` |

**`toplevel_next(+Pid, +Options?)`**

| Option | Values | Default |
|---|---|---|
| `limit` | integer $> 0$ | from `toplevel_call` |
| `target` | Pid | from `toplevel_call` |

### 3.3.4 Shell talking to a Prolog toplevel

Below, we show an example of how to create and interact with a toplevel process from a shell. We start by spawning a new toplevel, using `toplevel_spawn/2` with the `monitor` option to instruct the toplevel to send us a `down` message when the process eventually terminates. The `load_list` option is used to populate the private Prolog database belonging to the toplevel actor with two simple unit clauses `p(a)` and `p(b)`:

```
?- toplevel_spawn(Pid, [
       session(true),
       monitor(true),
       load_list([p(a),p(b)])
   ]).
Pid = 74122981.
?-
```

With this, the toplevel actor is initiated, and with the PTCP now in state **s1**, it is ready to accept messages and signals sent by the other `toplevel_*` predicates.

The `monitor` and `load_list` options are inherited from `spawn/1-3`. However, `toplevel_spawn/2` offers a couple of additional options that can be used to specify the behavior of toplevel actor at creation time. In the above example the `session` option is used to configure the toplevel to run a multi-goal session rather than exit after having run one goal to completion.

### Making the toplevel call a goal

Let us see what happens if we call `toplevel_call/2` with the default values for options:

```
?- toplevel_call($Pid, p(X)).
true.
?- flush.
Shell got success(74122981,[p(a),p(b)],false)
true.
?-
```

The answer is returned to the mailbox of the calling process in the form of an *answer term* with three arguments. The functor of the term represents its *type*. In this case, it shows that the goal succeeded. (In other cases it might indicate a failure or an error.) The first argument of the success term is the pid, and the list in the second argument represents the two solutions that were computed. The value `false` in the third argument of the term indicates that no more solutions exist.

Since the `session` option was set to true, the PTCP is now, after a brief visit to state **s2** for the execution of the goal, back in state **s1** and is prepared to accept another use of `toplevel_call/2-3`.

### Using the template option

Below, `toplevel_call/3` is called with the goal `p(X)` and with the `template` option set to the variable `X`:

```
?- toplevel_call($Pid, p(X), [
       template(X)
   ]).
true.
?- flush.
Shell got success(74122981,[a,b],false)
true.
?-
```

Note how the value of the `template` option determined the form of the list of solutions in the second argument of the answer term. The relation to the Prolog built-in standard predicate `findall/3` is evident.

### Using the `offset` and `limit` options

When querying a relational database using SQL or an RDF dataset using SPARQL, the use of the `OFFSET` and `LIMIT` keywords serves to control the subset of results that are returned by a query. In Web Prolog they can be used as options for the configuration of calls to `toplevel_call/3`. The `limit` option specifies the maximum number of solutions to return, while the `offset` option specifies the number of solutions to skip before starting to return solutions. Here is what happens if `limit` is set to `1`:

```
?- toplevel_call($Pid, p(X), [
       template(X),
       limit(1)
   ]).
true.
?- flush.
Shell got success(74122981,[a],true)
true.
?-
```

The value `true` in the `success` term means that the state machine is now in state **s3**, waiting for a message **next** or **stop** to appear. Calling `toplevel_next/1` produces the next solution:

```
?- toplevel_next($Pid).
true.
?- flush.
Shell got success(74122981,[b],false)
true.
?-
```

`toplevel_call/3` supports the `offset` option. Together with the `limit` option it allows the call to pick out an arbitrary slice of solutions to a goal, even when the number of solutions is infinite:[2]

```
?- toplevel_call($Pid, between(1,infinite,I), [
       offset(100),
       template(I),
       limit(3)
   ]).
true.
?- flush.
Shell got success(74122981,[101,102,103],true)
true.
?-
```

[2] It may seem odd and not all that useful for `toplevel_call/2-3` to support the `offset` option, but it turns out to be important when implementing the stateless HTTP protocol.

Calling `toplevel_next/1` produces three more solutions:

```
?- toplevel_next($Pid).
true.
?- flush.
Shell got success(74122981,[104,105,106],true)
true.
?-
```

Note that the value of the `limit` option set in the call to `toplevel_call/3` was still in effect. However, `toplevel_next/2` accepts the `limit` option too, which allows us to increase or decrease the limit on the number of solutions returned:

```
?- toplevel_next($Pid, [
       limit(5)
   ]).
true.
?- flush.
Shell got success(74122981,[107,108,109,110,111],true)
true.
?-
```

Calling `toplevel_stop/1` will stop the execution of the goal:

```
?- toplevel_stop($Pid).
true.
?-
```

These options are particularly useful in several scenarios: pagination of results in web applications, where it is impractical to retrieve and display all records at once; implementing infinite scrolling interfaces that load data dynamically as the user scrolls; handling goals with very many or infinitely many solutions without overwhelming the client; and reducing the number of network roundtrips on the Prolog Web by fetching solutions in batches. A special case is when `limit` is set to 1, which makes it straightforward to implement a Prolog-style REPL in JavaScript, where each press of `;` triggers a new request for the next solution.

### Using the `once` option

The `once` option only makes sense in combination with the `limit` option. Default is `false` but when set to `true`, the list of solutions in a `success` answer term are the only solutions that the call will produce (as indicated by `false` in the third argument of the answer term):

```
?- toplevel_call($Pid, between(1,infinite,I), [
       offset(100),
```

```
       template(I),
       limit(3),
       once(true)
   ]).
true.
?- flush.
Shell got success(23091243,[101,102,103],false)
true.
?-
```

### A toplevel can do a kind of I/O

Whenever a toplevel is in state **s2** and doing some real work, it is able to send messages to its parent using the ordinary send operator. However, a better idea can often be to use `output/1`:

```
?- toplevel_call($Pid, output(hello)).
true.
?- flush.
Shell got output(74122981,hello)
Shell got success(74122981,[output(hello)],false)
true.
?-
```

Input can be collected by calling `input/2`, which sends a `prompt` message to the client, which in turn can respond by calling `respond/2`:

```
?- toplevel_call($Pid, input('Input', X)),
   receive({Answer -> true}).
Answer = prompt(74122981,'Input').
?- respond($Pid, hello),
   receive({Answer -> true}).
Answer = success(74122981,[input('Input',hello)],false).
?-
```

### Aborting a nonterminating goal

Our toplevel actor is still not dead so let us see what happens when we ask the toplevel to first update its private Prolog database with a silly recursive clause for a predicate `p/0` and then call it:

```
?- toplevel_call($Pid, assert((p :- p))),
   receive({Answer -> true}).
Answer = success(74122981,[assert((p:-p))],false)
?- toplevel_call($Pid, p).
```

```
true.
?-
```

Although nothing is shown in the terminal, it is clear that the toplevel is now just wasting CPU cycles to no avail. Fortunately, a nonterminating goal can be aborted by calling `toplevel_abort/1`:

```
?- toplevel_abort($Pid).
true.
?-
```

With this, the PTCP is back in state **s1**.

### Exiting the toplevel

When we are done talking to the toplevel we can kill it. As the `monitor` option was set to `true`, we should expect to receive a `down` message:

```
?- exit($Pid, goodbye),
   receive({Answer -> true}).
Answer = down(74122981,74122981,goodbye).
?-
```

### Toplevels and the message deferring mechanism

In the following example, a toplevel is spawned, and then `toplevel_next/1` is called. The protocol in state **s1**, which is obviously not in a state where it can react on the **next** message (see Figure 3.3). The message is therefore deferred and it is not until `toplevel_call/3` is called and the protocol changes states that it has an effect.

```
?- toplevel_spawn(Pid).
Pid = 78340943.
?- toplevel_next($Pid).
true.
?- flush.
true.
?- toplevel_call($Pid, member(X,[a,b,c]), [
       limit(1),
       template(X)
   ]).
true.
?- flush.
Shell got success(78340943, [a], true)
Shell got success(78340943, [b], true)
```

```
true.
?-
```

Note that messages sent to a toplevel will often be handled in the right order even if they arrive in the "wrong" order (e.g. **next** before **call**). This is due to the selective receive which defers their handling until the PTCP permits it. The messages **abort** and **exit**, however, will never be deferred. This is guaranteed by the fact that **abort** and **exit** are valid transitions from any state in the protocol (see Figure 3.3). (They are actually signals rather than messages).

One way to think of this is in terms of the so called *robustness principle*: "Be conservative in what you send, be liberal in what you accept."[3] Due to the deferring behavior a toplevel is liberal in this way, but, as implied by the principle, clients are advised not to rely on this behavior.

### Redirecting answers

The `target` option can be used to redirect the answer terms produced by the toplevel to any actor of choice. By default, it is set to the pid of the parent but allows us to instruct a toplevel actor to send answer terms to a different destination. To demonstrate how this works, we first create a simple actor that can serve as a target:

```
?- spawn(( repeat,
           receive({
               Msg ->
                   format("Received ~p~n", [Msg]),
                   fail
           })
         ), Pid0).
Pid0 = 98380209.
?-
```

The pid of that actor can now be used as the value of the target option:

```
?- toplevel_spawn(Pid, [
       target($Pid0)
   ]).
Pid = 97919106.
?-
```

When `toplevel_call/3` is called, the `success` answer term is printed by the designated target actor:

```
?- toplevel_call($Pid, between(1,1000,N), [
       template(N),
       limit(5)
```

[3] `https://en.wikipedia.org/wiki/Robustness_principle`

```
   ]).
true.
Received success(97919106,[1,2,3,4,5],true)
?-
```

So, 98380209 rather than the shell received the message from 97919106.

Calling `flush/0` shows that the mailbox belonging to the shell is empty:

```
?- flush.
true.
?-
```

The `target` option is supported by `toplevel_spawn/2`, `toplevel_call/3` and `toplevel_next/2`. It is not supported by the other `toplevel_*` predicates as they do not produce answer terms. It can be used in calls to `output/2`. See Appendix A for more details.

### 3.3.5 Programming with toplevel actors

The toplevel actor is an important kind of agent in the Prolog Trinity ecosystem. However, the reasons for its importance will have to be deferred to later chapters, and this section only offers a few exercises to build familiarity with it.

#### A synchronous predicate API to toplevels

The communication between a client and a toplevel illustrated above is asynchronous, but if we want, it is quite easy to define a synchronous alternative using a technique we have already seen. A predicate `wait_for_answer/3` can be defined like so:

```
wait_for_answer(Pid, Answer, Options) :-
    receive({
        Answer if arg(1, Answer, Pid) ->
            true
    }, Options).
```

Then `toplevel_call_synch/3-4` can be implemented as follows:

```
toplevel_call_synch(Pid, Goal, Answer) :-
    toplevel_call_synch(Pid, Goal, Answer, []).

toplevel_call_synch(Pid, Goal, Answer, Options) :-
    toplevel_call(Pid, Goal, Options),
    wait_for_answer(Pid, Answer, Options).
```

and `toplevel_next_synch/2-3` like so:

```
toplevel_next_synch(Pid, Answer) :-
    toplevel_next_synch(Pid, Answer, []).

toplevel_next_synch(Pid, Answer, Options) :-
    toplevel_next(Pid, Options),
    wait_for_answer(Pid, Answer, Options).
```

Note that since `toplevel_stop/1` does not produce a response message, no `toplevel_stop_synch/2-3` is defined, and `toplevel_stop/1` can be used as it is.

Here is a simple test that shows how it works:

```
?- toplevel_spawn(Pid).
Pid = 67590967.
?- toplevel_call_synch($Pid, mortal(Who), Answer, [
       template(Who),
       limit(1)
   ]).
Answer = success(67590967, [socrates], true).
?- toplevel_next_synch($Pid, Answer).
Answer = success(67590967, [plato], true).
?- toplevel_next_synch($Pid, Answer).
Answer = success(67590967, [aristotle], false).
?-
```

### Reconstructing findall/3 and findnsols/4

In order to specify the semantics of a new Prolog predicate API it can be a good idea to implement built-in Prolog predicates with a well-known semantics using the API. For example. if we did not already have a built-in predicate `findall/3`, using a toplevel actor we could have implemented it like so:

```
findall(Template, Goal, List) :-
    toplevel_spawn(Pid, [
        session(false)
    ]),
    toplevel_call(Pid, Goal, [
        template(Template)
    ]),
    receive({
        success(Pid, List, false) ->
            true ;
        failure(Pid) ->
            List = [] ;
        error(Pid, Error) ->
```

```
            throw(Error)
    }).
```

It may not be useful, but it does say something about the relation between `findall/3` and the `toplevel_*` predicates.[4] Of course, implementing these predicates in separate processes makes absolutely no sense. Examples such as these only exercises some of the capabilities of the `toplevel_* API` and may seem somewhat pointless. Concurrency and distribution is where they really shine.

We can take this one step further by also implementing `findnsols/4`, which works like `findall/3`, but generates at most N solutions to the specified goal. This predicate is especially useful if the goal may have an infinite number of solutions.

```
findnsols(N, Template, Goal, List) :-
    toplevel_spawn(Pid, [
        session(false)
    ]),
    toplevel_call(Pid, Goal, [
        template(Template),
        limit(N)
    ]),
    wait_solutions(Pid, List).

wait_solutions(Pid, List) :-
    receive({
        success(Pid, List, false) ->
            true ;
        success(Pid, List0, true) ->
            (   List = List0
            ;   toplevel_next(Pid),
                wait_solutions(Pid, List)
            ) ;
        failure(Pid) ->
            List = [] ;
        error(Pid, Error) ->
            throw(Error)
    }).
```

Ciao Prolog provides this predicate, but only in a deterministic version. The version provided by SWI-Prolog is nondeterministic, and so is our reconstruction. We borrow an example of its use from the SWI-Prolog manual:[5]

```
?- findnsols(5, I, between(1, 12, I), L).
L = [1, 2, 3, 4, 5] ;
L = [6, 7, 8, 9, 10] ;
```

[4] Here, we were inspired by (Tarau, 2000).

[5] `https://www.swi-prolog.org/pldoc/doc_for?object=findnsols/4`

```
L = [11, 12].
?-
```

Interestingly, it turns out that the version provided by SWI-Prolog serves as an important building block in our proof-of-concept implementation of the `toplevel_*` predicates. (It was implemented in SWI-Prolog by Jan Wielemaker in order to support `library(pengines)`, the library on top of which SWISH is built.)

## 3.4 Prolog nodes are also agents

As is evident from our agent taxonomy in Figure **??**, we regard a Prolog node as a kind of Prolog agent. We consider it as such because, as we shall see, a node incorporates a significant subset of the capabilities of a toplevel actor. In particular, it enables clients to submit queries and to receive solutions lazily – one at a time (or, if required, more than one at a time), in the order that Prolog finds them.

The diagram in Figure 3.4 illustrates the anatomy of an executing Prolog node, which is accessed by a user-client that knows its URI and is authorized to utilize its capabilities.

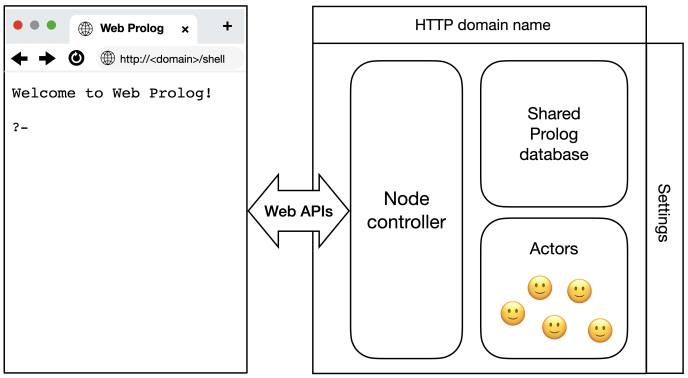



**Fig. 3.4** The anatomy of a Prolog node. A terminal running in a web browser is attached to it.

As suggested by the diagram, a node is comprised of components such as a node controller, web APIs, a shared database containing predicates defined by the owner, a container serving as the "home" of actors such as echo actors, supervisors and toplevels, and various settings for the owner to configure the node. As we shall see, not all nodes include all these components.

### 3.4.1 The node controller

In the most capable profile, the node controller is entrusted with a multitude of responsibilities designed to ensure the smooth and secure operation of the node. Although not every node controller undertakes all of these tasks, the most advanced implementations are expected to manage web API connections, authenticate and authorize clients, and spawn actors upon client request. Additionally, they impose various resource restrictions to maintain optimal performance, oversee registries, monitors, and links, and guarantee that messages sent back to clients adhere to the required format. The node controller also terminates actors that have lost client connections and implements shell utilities such as the $Var substitution mechanism and the redefinition of I/O predicates.

From an external perspective and when dealing with actors, the intricate workings of the node controller are intended to remain largely concealed, functioning much like an invisible "ether" through which messages seamlessly flow. In this manner, the node controller abstracts away the underlying technical details, allowing users to benefit from its comprehensive management capabilities without the need to engage directly with its multifaceted processes.

### 3.4.2 The shared Prolog database

A node may host a shared Prolog database accessible over HTTP. It can be:

- *self-contained* - all predicates required for the exported relations are present locally, making the dataset directly consumable by other nodes or clients;
- *dependent* - code or data rely on external modules or remote predicates, which reduces portability unless those dependencies are resolvable at the destination.

The predicates – which will be referred to as the *node-resident* predicates – are typically maintained by the owner of the node. External users are not able to make any changes to them, only the owner of the node is allowed to do so. Any client connected with the node has access to these predicates in addition to the Web Prolog built-in predicates, and can pose queries over the web APIs formulated in terms of them.

The contents of the file `mortal.pl` must be valid Web Prolog source code. It may be something like this:

```
mortal(Who) :- human(Who).

human(socrates).
human(plato).
human(aristotle).
```

This is also what a user is presented with when steering an ordinary browser to the URI `http://n7.org`. This is an example of a shared Prolog database that is

*self-contained*. In Chapter 4 we will demonstrate how this property makes it possible to achieve what we think of as *instant portability*.

However, a file might also look like this:

```
:- use_module(philosophers, [philosopher/1]).

mortal(Who) :- human(Who).

human(socrates).
human(Who) :- philosopher(Who).
```

Since the clauses for `philosopher/1` are not available, this database is *dependent*.

### 3.4.3 Deploying a node

To make this as concrete as possible, let us assume that the owner of a node wants to make the data in the file `mortal.pl` available to clients. The owner uses the following command to set up a Prolog node:

```
$ prolog-node --port=80 --src=mortal.pl --settings=settings.pl
```

As a result, a new node is created and the contents of the file is loaded into its shared Prolog database. The node is configured according to the settings given in the file `settings.pl`. After having performed all the usual moves necessary for the deployment of a web server, access to the node is offered through the set of API endpoints listed in Table 3.1:

| **API endpoint** | **Resource or functionality** |
|---|---|
| `http://n7.org` | The shared Prolog database |
| `http://n7.org/shell` | A simple shell environment |
| `http://n7.org/call` | The stateless HTTP API |
| `http://n7.org/toplevel_spawn` | The semi-stateful HTTP API |
| `ws://n7.org/actor` | The stateful WebSocket API |

**Table 3.1** Proposal for how to link URIs with targets. Note that the stateful and semi-stateful APIs have additional endpoints.

By steering an ordinary browser to the URI `http://n7.org` an authorized user may want to just browse the program. Or perhaps query it by making a visit to `http://n7.org/shell`. Or perhaps the plan is to build an end-user application talking to the node over any of its web APIs.

Note also that because a Prolog node is also a web server, it may offer other HTTP endpoints as well, some of which may not be relevant to its core functionality.

Other ecosystems show that this strategy works. In the Prolog world, Jan Wielemaker's *Cliopatria* demonstrated how a substantial Web application could be delivered as a ready-to-run system: users provided their data, made modest extensions, and obtained a full Semantic Web portal. In the Elixir world, Phoenix goes even further by generating complete, fault-tolerant applications with supervision, telemetry, routing, and deployment already in place. Developers simply add the behaviour that makes the system theirs. These examples illustrate how a language ecosystem can supply standardised, reproducible scaffolds that capture the best practices of a community.

### 3.4.4 Settings

As suggested by Figure 3.4, a Web Prolog node is governed by a set of *owner-controlled settings*. Only the node owner may change these settings; some of them may be *readable* by clients over HTTP, so that clients can discover the node's advertised capabilities and operational limits. Collectively, the settings determine which capabilities the node offers, who may access them, and the resource envelope within which actors and sessions are allowed to run.

It is useful to group settings into five categories:

1. **Identity and advertised capability.** The node advertises a *profile* to clients. This is primarily a discovery and compatibility mechanism: it tells clients what kind of service they can expect.
2. **Resource governance.** The node may enforce limits on injected code size, actor-local database size, cache bounds, and the maximum number of concurrent actors that client code is allowed to spawn. At the transport layer it may also enforce connection-related limits (for example, the maximum number of connections, and the maximum number of agents sharing a connection). These limits are part of the node's resource policy and are set by the node owner.
3. **Admission control.** The node may restrict which clients may connect and which goals may be executed, for instance via allow/deny lists and related authorization rules.
4. **Session lifecycle and PTCP timeouts.** Two timeouts are particularly important for PTCP sessions (cf. the PTCP state diagram in Figure 3.3). A *running timeout* bounds how long a session may remain in the working state (i.e **s2**). If this timeout is exceeded, the node aborts the current goal and returns an error (e.g. `timeout_exceeded`) to the caller, while keeping the session actor alive. An *idle timeout* bounds how long a session may remain inactive in an idle state (i.e **s1** or **s3**). If this timeout is exceeded, the node terminates the session actor. If the session is monitored, the client will subsequently observe a `down` message consistent with normal actor termination.
5. **Bundle loading policy.** The node may require, or merely prefer, that loaded bundles are self-contained, i.e. do not rely on unresolved external dependencies.

Note that readable over HTTP does not mean modifiable: the intention is that the node may expose a read-only subset of the settings that are relevant to interoperability and capacity planning, while keeping sensitive policy (notably admission rules) private.

Several of these settings are best understood as *capability gates* rather than mere tuning parameters. In particular, a node distinguishes between what the owner may do and what external clients may do. Clients are intended to program *session-local* concurrent computations, while the owner configures the long-lived surface of the node and installs durable services. This distinction will later be reflected in which concurrency-oriented predicates and options are admitted for client code, especially for lifetime control (for example whether a spawned actor may outlive its parent) and, in the distributed case, which combinations of remote placement and lifetime options are permitted.

### 3.4.5 Two transports – three web APIs

The primary responsibility of the node controller is to facilitate web API interactions. The most capable nodes are designed to support two different transports – HTTP and WebSocket – and three distinct API types: a stateless HTTP API, a semi-stateful HTTP API, and a stateful WebSocket API. This architecture enhances flexibility by increasing the likelihood that at least one API will effectively serve a given end-user application.[6]

#### The HTTP transport

The *Hypertext Transfer Protocol* (HTTP) is a core communication protocol of the World Wide Web, enabling the exchange of resources like web pages and data between clients and servers. Operating as a stateless request–response protocol, it supports methods such as GET, used to retrieve resources, and POST, used to submit data to a server.

A client interacting with a node by making requests over a HTTP connection is illustrated in Figure 3.5.


**Fig. 3.5** Interaction between a client and a node over an HTTP connection.

[6] Strictly speaking, HTTP and WebSockets both live in the application layer, not the transport layer. But in practice, even if TCP is the real transport protocol under the hood, it makes sense to think of them as alternative ways of transporting data between agents.

HTTP typically uses TCP for reliable transmission and supports secure communication through HTTPS with TLS encryption. Evolving since the 1990s, HTTP has advanced through versions like HTTP/2 and HTTP/3, enhancing performance and features like caching, authentication, and content negotiation. There is a vast infrastructure for HTTP and HTTPS that already exists (proxies, firewalls, caches, and other intermediaries), ensuring that security requirements of the modern web are fulfilled and the browser and the node can validate each other.

### The WebSocket transport

In order to enable a client to control all aspects of a set of toplevels or other kinds of actors, the most capable type of node offers a WebSocket sub-protocol. WebSocket is a real-time, low latency, bidirectional protocol for asynchronous communication between a client and a node. A WebSocket sub-protocol is an application-layer protocol, which operates on top of the WebSocket protocol, leveraging the transport services provided by TCP. The three phases in the life-cycle of a WebSocket connection is illustrated in Figure 3.6.

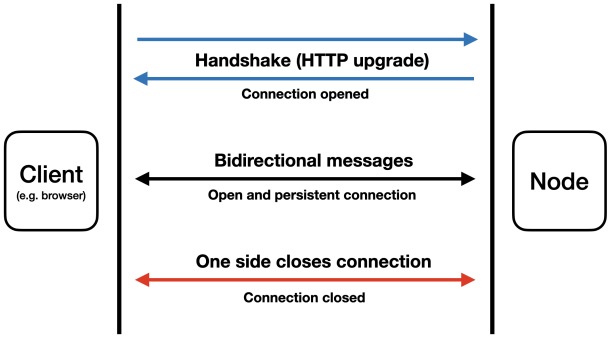



**Fig. 3.6** The life-cycle of an interaction between a client and a node over a WebSocket connection.

In the first phase (in blue in the diagram) a WebSocket connection is initiated by the client via an HTTP connection which is then switched over to the WebSocket protocol. In the second phase (in black), the client and the node starts talking to each other. This is where the sub-protocol comes in.

Subject to the value of a HTTP header or parameter, answers and other kinds of messages arriving from the node are returned in the form of either JSON or Prolog text. Client code written in languages other than Prolog would normally request that they be encoded in JSON.

WebSocket client libraries are available for the most common general programming languages. However, the most common use of WebSockets is in communication between a web browser and a server. In all the major web browsers, the WebSocket

object provides a JavaScript API for creating and managing a WebSocket connection to a server, as well as for sending and receiving data on the connection.[7] The role of JavaScript here is to create a new websocket, to define the necessary handlers for `onopen`, `onmessage`, `onerror` and `onclose` messages, and to call the methods `send` or `close` for sending messages or closing the connection.[8]

In the next section, and with the help of a number of scenarios, the actual APIs will be demonstrated. For details, see Appendix A.

## 3.5 Node profiles

Not all Prolog nodes are equal. They vary in the capabilities they offer clients, and we use the term *profile* – borrowed in the standards sense – for the declared set of capabilities a node commits to providing. A profile is a tailored subset of the full design: it constrains what a node must offer in order to make that offer verifiable, portable, and predictable. Less capable profiles are easier to specify, implement, and standardise, while still participating in the same ecosystem.[9]

We distinguish two kinds of capability: *language capabilities* (which Prolog fragments and predicates are available) and *interactional capabilities* (which transports and web APIs the node supports). Four profiles are defined: RELATION, ISOBASE, ISOTOPE, and ACTOR. Their nesting is shown in Figure 3.7: features in the inner rings are required by every profile, while features in the outer rings are restricted to more capable ones. For example, owner-defined shared predicates and the stateless HTTP API appear inside the RELATION ring, signifying that these features are supported by all profiles. The WebSocket API appears in the outermost ring, indicating that it is guaranteed to be available only in the ACTOR profile.

The RELATION profile is the base. It requires only the stateless HTTP API and a set of owner-defined predicates; it does not require any Web Prolog built-in predicates. ISOBASE adds the standard built-in predicate set, `rpc/2-3`, and asynchronous RPC. ISOTOPE further adds a session-oriented toplevel actor accessible over the semi-stateful HTTP API, enabling persistent state and interactive I/O within a session. ACTOR adds the full suite of Erlang-style concurrency predicates – `spawn/2-3`, `!/2`, `receive/1-2`, and the derived predicates built on them – and exposes them over a bidirectional WebSocket connection. A node offering the ACTOR or ISOTOPE profile can be both queried and programmed; a node offering only ISOBASE or RELATION can only be queried.

---

[7] See `https://www.w3.org/TR/websockets/`

[8] For a good general introduction to WebSocket we recommend (Lombardi, 2015).

[9] Throughout this section *capability* means a client-visible feature or power provided by a node – support for a given web API or built-in predicate. The narrower capability-security sense of an unforgeable authority token – a value whose possession confers the ability to act – is developed in Chapter 4. The two uses are related: node-level capabilities determine what actions are available in principle, while capability values (such as pids) determine which of those actions a client can actually perform.

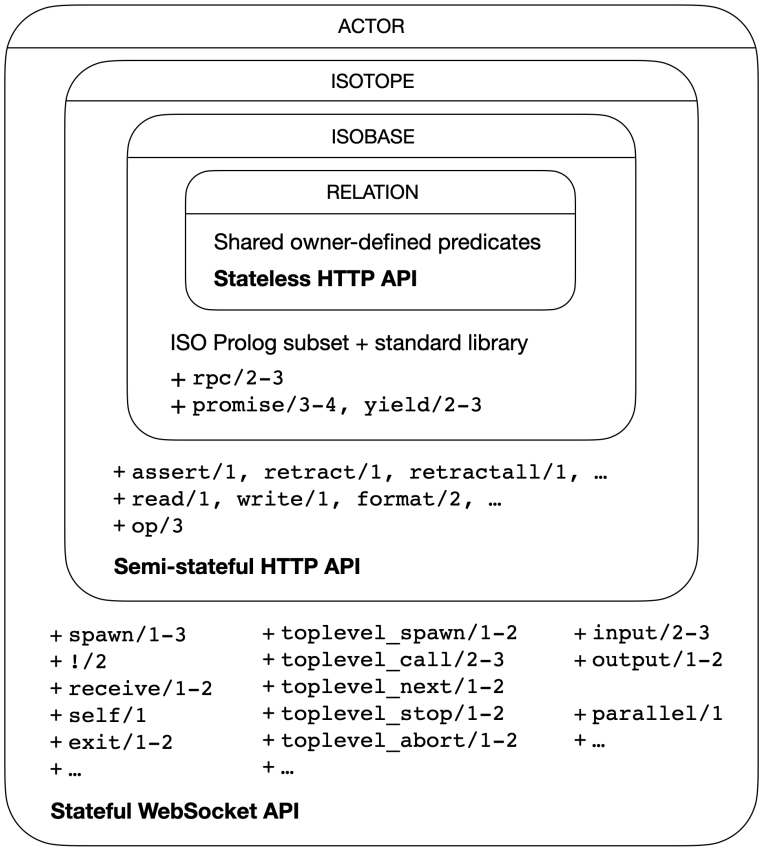



**Fig. 3.7** The profile hierarchy. Each ring adds capabilities to those of the profiles inside it.

The RELATION profile differs from the others in kind, not merely in degree. An ISOBASE, ISOTOPE, or ACTOR node accepts arbitrary Prolog goals – within the predicate subset defined by the profile – and evaluates them. A RELATION node does not. It exposes specific named relations through a fixed query interface: a client can ask `wife(H,W)` because the node advertises that relation, not because it evaluates arbitrary Prolog code. The goal in a request to a RELATION node is better understood as a pattern matched against a known schema than as a program to execute. A conformant RELATION node need not contain a Prolog engine at all.

### Conformance

To be considered *conformant*, a node *must* provide clients with *all* capabilities specified by its profile. For instance, an ACTOR node *must* offer clients access to

the three web APIs and the entire set of built-in predicates indicated in Figure 3.7. Similarly, an ISOTOPE node *must* provide clients with the two HTTP APIs and all predicates defined in the ISOTOPE subset of Web Prolog. Other profiles follow analogous requirements.

A node *may* provide *more* than what is mandated by its profile, including additional predicates or web APIs. For example, a RELATION node may support the `rpc/2-3` predicate as well as both the stateless and semi-stateful HTTP APIs, but it is not obligated to do so. Only the stateless API is required. Clients that depend on such extras sacrifice portability across nodes at that profile level.

It is perfectly reasonable for a node to not be conformant to any profile whatsoever. For example, a node whose only purpose is to serve a fast-paced, web-based game may make good use of the WebSocket API, but its owner may choose to block the use of everything that is not required for the support of the game. Such a node makes no portability claim at the level of interaction; it is the act of conforming to a profile that brings a node into the scope of the portability guarantees defined in Section 9.2.

The hierarchy is intentionally non-exhaustive. One can also imagine *orthogonal* profiles that cut across the capability ladder by adding restrictions or guarantees rather than extending it – a PURE profile limiting clients to declarative Prolog, for instance, or a statechart discipline as an ACTOR sub-profile. Section 3.5.6 returns to this possibility.

### What are profiles good for?

Profiles do two things simultaneously: they give clients a stable contract and give node implementers a scalable commitment. A client talking to an ACTOR node knows it may spawn actors, use persistent sessions, and receive push events over WebSocket; the same client talking to an ISOBASE node knows it may not. A node implementer choosing to deploy at the RELATION level knows that a Python microservice wrapping a database is sufficient; one targeting ACTOR knows what the full engineering cost entails. The profile a node advertises is thus a public, verifiable statement of what both parties can assume.

Crucially, this contract operates at the level of *observable interaction*, not implementation. Across all four profiles, the same shell-style pattern is preserved: a client submits a goal and receives solutions one at a time, lazily, on demand. What differs is only the infrastructure behind that pattern – stateless HTTP, a semi-stateful session actor, or a live WebSocket stream. In the extreme case a RELATION node may produce correct answers without any Prolog execution at all; from the client's perspective the interaction is indistinguishable. This separation of observable contract from implementation commitment is what allows the profile hierarchy to be both strict (conformance is testable) and open (any conformant implementation qualifies, regardless of how it works internally).

The shell-style interaction pattern deserves emphasis here because it is central to logic programming itself. Presenting one solution at a time, pausing for user input, and resuming search on demand exposes the mechanics of backtracking and

makes nondeterministic computation tractable and inspectable. Profiles ensure that this familiar interaction can be offered even by nodes with limited capabilities, while allowing more capable nodes to support richer forms of control, state, and concurrency.

Profiles also carry a security dimension, developed fully in Section 4.5. A node's profile determines not only what a client can *do* but what it can *express*: excluding I/O and dynamic database modification from ISOBASE is not merely a protocol convenience but removes those operations from the client-accessible language fragment entirely. Greater expressiveness at higher profiles therefore comes paired with greater exposure, a tradeoff the security architecture addresses through inexpressibility, name discipline, and lifetime containment. The profile a node advertises is, in this sense, also a declaration of its security posture.

The following subsections work through five scenarios – RELATION, ISOBASE, ISOTOPE, ACTOR via browser, and ACTOR via social robot – each illustrating the interaction model that its profile enables.

### 3.5.1 Browser talking to a RELATION node

As indicated by the profile hierarchy in Figure 3.7, the stateless HTTP API is the only web API that *must* be supported by a RELATION node. This does not distinguish it from nodes having a more capable profile, as this is required by them too. What makes it stand out is that it is not required to offer access to any Web Prolog built-in predicates or control constructs *at all*. Even control constructs such as `,/2` and `;/2`, may not be available.

In the scenario depicted in Figure 3.8, a user equipped with an ordinary web browser is talking to the RELATION node at `http://n1.org`. The web-based terminal displays the remnants of a session during which the user executed two queries. The two question marks signify that neither the owner-defined resources nor the way answers are produced is known by the user.
The interaction started with the loading of the Prolog shell at `http://n1.org/shell`. The interaction opened with the welcome banner and a `?-` prompt; a JavaScript loop then began reading queries, translating them into HTTP requests, and writing answers back to the terminal.

Let us look at how the first solution to `?-wife(H,W)` was retrieved by using the stateless HTTP API. The URI pointing to this solution is:

```
http://n1.org/call?goal=wife(H,W)&limit=1&offset=0
```

Note the use of `limit` and `offset` parameters. Although here they are parameters rather than options, their defaults and the effect that they have on answer terms is the same as for the options accepted by `toplevel_call/3`.

URIs such as these are simple, they are meaningful, they are declarative, they can be bookmarked, and responses are cacheable by intermediates. Most of these

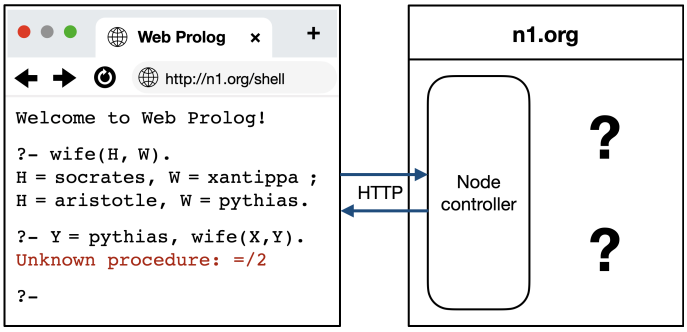



**Fig. 3.8** A scenario involving a web browser in conversation over HTTP with a remote RELATION node. The two question marks signify that neither the owner-defined resources nor the way answers are produced is known by the user.

desirable properties are there because the client-node interaction involved during the process of resolving the URIs is stateless.

When the user entered the query `?-wife(H,W)`, (and assuming that a relative path was used instead of an absolute one) the following GET request was made to the node:

```
GET /call?goal=wife(H,W)&limit=1&offset=0 HTTP/1.1
Accept: application/json
```

The `Accept` header requested responses in JSON. (In this case the header could have been left out since JSON is the default.)[10]

The node responded with the following JSON structure, indicating that the query succeeded:

```
{ "type": "success",
  "data": [{"H":"socrates","W":"xantippa"}],
  "more": true }
```

The `data` field contains the variable bindings and the value `true` of the `more` field signals that further solutions may be available. This triggered the printing of the variable bindings and served as a prompt for how to continue the exploration.

The user wanted to know if there were more solutions and typed a semicolon which led to the following request being made, looking exactly like the previous request, except that the `offset` parameter was incremented from `0` to `1`:

```
GET /call?goal=wife(H,W)&limit=1&offset=1 HTTP/1.1
```

The node responded like so:

[10] A `format` parameter is also supported as this allows us to easily make a request from the address field of a web browser.

```
{ "type": "success",
  "data": [{"H":"aristotle","W":"pythias"}],
  "more": false }
```

The `more` field was now `false`, indicating that no further solutions existed, and after having printed the next set of bindings and a full stop, the shell again produced the `?-` prompt, inviting the user to enter another query.

At this stage of the interaction, it is reasonable for the user to expect that `wife/2` has the mode `wife(?,?)` and that queries such as `?-wife(socrates,W)` or `?-wife(H,pythias)` should also succeed. A query such as `?-wife(plato,W)` should fail, however, and the response would be the following:

```
{ "type": "failure" }
```

Finally, an erroneous query, such the attempt in the scenario to query a predicate which is not defined, should raise an error, and show up as a response like so:

```
{ "type": "error",
  "data": "Unknown procedure: =/2" }
```

We can summarize this by saying that a RELATION node must offer clients a truly relational query interface over stateless HTTP. Any node failing to meet this expectation is not a conformant RELATION node. A RELATION node is required to respond to a client as if it were Prolog, thereby allowing the user to lazily backtrack over possible solutions. Although a RELATION node may not truly perform backtracking, it must behave externally as though it does.

The lack of built-in predicates naturally makes a RELATION node very easy to implement. There are two quite different ways a RELATION node can come into existence. First, a node that already implements a richer profile can be turned into a RELATION node by installing a filter that inspects the goal parameter in each `/call` request and only allows it through if it matches a declared relation. Such a filter can be as simple as matching the submitted goal against a whitelist of terms – for instance, accepting only `employee(_,_,_)` and `name(_,_)`, and rejecting everything else with an error. A node that is supposed to serve a particular application rather than act as a general compute server will often operate this way: it implements ISOBASE or more internally, but presents a RELATION interface to its clients.

Second, a RELATION node can be implemented from scratch in any programming language. (Recall that we do not require that a Prolog node is written in Prolog, only that it can talk Prolog.) At a minimum, an implementation must parse Prolog terms and produce variable bindings for queries against its data. A Python web service backed by a relational database, or a microservice wrapping a key–value store, can present itself as a conformant RELATION node as long as it implements the stateless HTTP API and responds with the expected JSON structure. This is the profile that makes the Prolog Web interoperable with the rest of the Web: any structured data endpoint can, in principle, participate in the ecosystem.

### 3.5.2 Browser talking to an ISOBASE node

For a node to adhere to the requirements of the ISOBASE profile, it *must* support the same stateless HTTP API as required by the RELATION profile. In our next scenario, illustrated in Figure 3.9, the user is running the same query as in the previous scenario. It means that we do not need to describe the details of requests and responses again. Instead, we focus on the language capabilities of the node and the contents of the shared Prolog database offered by it, and visible in the figure.

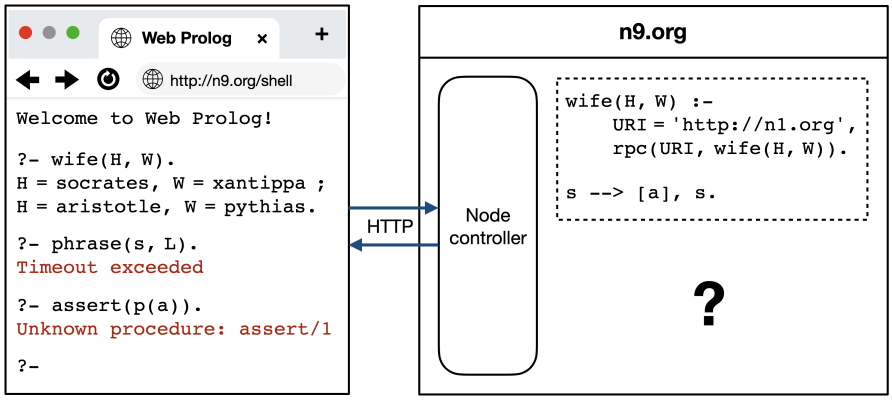



**Fig. 3.9** Stateless interaction in a scenario involving a web browser and a remote node.

The main difference compared to the previous scenario is that an ISOBASE node *must* make a fairly large number of Web Prolog built-in predicates available, including a large subset of ISO Prolog predicates (recall the cloud of ISO Prolog predicates in Chapter 2), standard library predicates, definite clause grammars, as well as predicates for working with the Web. (The scenario differs from the previous one in that it allows the user to browse the source code defining the predicates offered by the owner of the node.)

An ISOBASE node must also support `rpc/2-3`, an easy-to-use predicate for making *nondeterministic remote procedure calls* from one node to another over stateless HTTP, communicating in a way that is essentially *synchronous*. In the scenario in Figure 3.9, the URI `http://n1.org` in the first argument of the call to `rpc/2` shows that it accesses the relation `wife/2` made available by the node `n1.org` in the previous scenario.

In addition to the synchronous rpc/2-3, an ISOBASE node must also support *asynchronous* RPC using two built-in predicates; `promise/3-4` and `yield/1-2`. It turns out that these three predicates can all be implemented on top of the stateless HTTP API. They will be dealt with in more detail in Chapter 4.

Stateless interaction has one significant constraint: once a query is submitted, neither the client nor any other party can abort it. Because there is no control channel back to the running computation, ISOBASE nodes must enforce a timeout. This is what happened with the user's second query in the Figure 3.9 scenario: it was met with a `Timeout exceeded` message. Browsing the contents of the node's shared database, the problem seems to be that the node owner forgot to add a DCG rule `s --> []` that would have allowed the call to terminate.

Another limitation is that a truly stateless request-response protocol simply cannot support updates to the dynamic private database of an actor, nor I/O. It means that ISO predicates for modifying the dynamic database (e.g. `assert/1` and `retract/1`), or I/O (e.g. `read/1` and `write/1`), are *not* available. This is what happened with the user's third query, which was met with an `Unknown procedure` error message when trying to assert the clause `p(a)`. The predicate `assert/1` is simply not available to a client talking to an ISOBASE node.

### 3.5.3 Browser talking to an ISOTOPE node

An ISOTOPE node offers clients a shell backed by a toplevel actor. (Importantly, this does not mean that the internal `toplevel_*` predicates are directly exposed to programmers.) By using the process identifier of this actor as a session key in otherwise stateless HTTP requests, the node provides a semi-stateful API: each request is stateless in itself, but thanks to the pid they address the same long-lived toplevel actor.

The client's initial request can populate the actor's private database, which can then be updated dynamically throughout the session. Furthermore, although the client must initiate all interactions, the toplevel actor can not only respond to queries but can also prompt the client for further input or produce output at arbitrary moments. As long as the client keeps supplying the session identifier, this will work just fine.

In the scenario in Figure 3.10, the client established communication with a toplevel actor on an ISOTOPE node, which is a node that must provide all the capabilities that an ISOBASE node has, and then some.

The request below instructed the node controller to spawn a toplevel actor running as a session, and to populate its private database with the value of the `load_text` option. (Let us assume that the clause `q(X) :- p(X)` was picked up from an editor window that is not shown in the diagram.)

```
POST /toplevel_spawn HTTP/1.1
Content-Type: application/json

{ "options": "[session(true), load_text('q(X):-p(X).')]" }
```

The node responded with a JSON structure containing the pid for the spawned toplevel actor:

```
{ "type": "spawned",
```

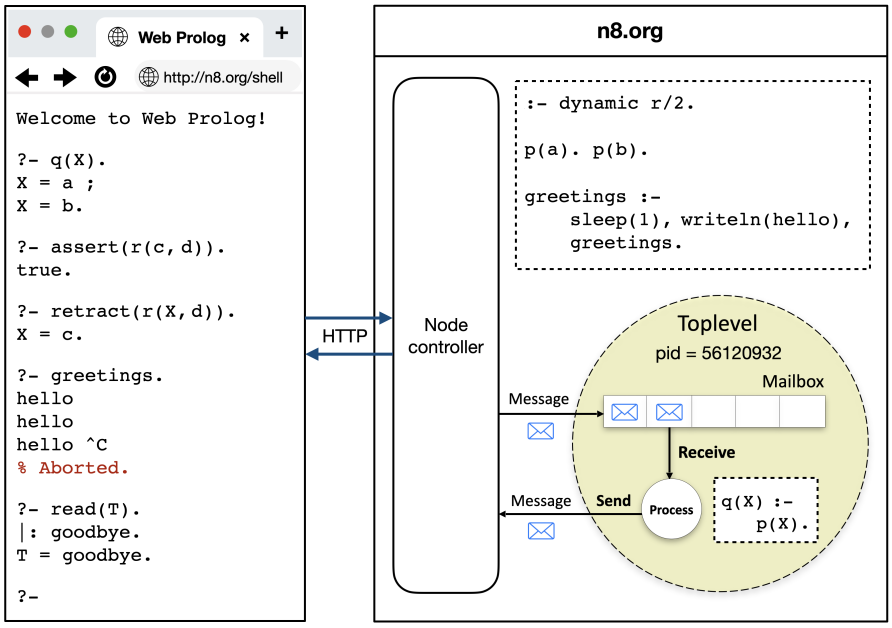



**Fig. 3.10** A scenario involving a web browser in conversation over semi-stateful HTTP with a remote ISOTOPE node running a toplevel actor.

```
  "pid" : 56120932 }
```

This produced the prompt `?-` in the terminal, inviting the user to enter a query. The query `?-q(X)` caused the JavaScript shell to make the following request to the node:

```
GET /toplevel_call?pid=56120932&goal=q(X)&limit=1 HTTP/1.1
```

(Note that the pid must be sent along when querying the toplevel.) The request told the node controller to make a call to `toplevel_call/3` and to convert the resulting answer term into a JSON structure that encodes the first solution:

```
{ "type": "success",
  "pid" : 56120932,
  "data": [{"X":"a"}],
  "more": true }
```

Except for the pid, the JSON structure representing this solution is identical to the response when making a stateless request.

The value `true` of the `more` property signifies that more solutions may be available. Asking for the next solution results in a similar message, except that the value of the `data` property becomes `[{"X":"b"}]` and the value of `more` becomes `false`.

When the user asked for another solution, the following request was made:

```
GET /toplevel_next?pid=56120932 HTTP/1.1
```

Note that it took the *combination* of the clause that was loaded into the toplevel's private Prolog database, and data in the shared database to answer the query `?-q(X)`.

Because the toplevel persists across requests, it makes sense to allow the client to update its private database using `assert/1` and `retract/1` during the session.

```
GET /toplevel_call?pid=56120932&goal=assert(r(c,d)) HTTP/1.1
```

With this, the clause `r(c,d)` ended up in the private database of `56120932`, but was removed again when the user decided to call `retract(r(X,d))`.

The user then called `greetings/0`, which calls `sleep(1)` and `writeln(hello)` in a loop. The `writeln/1` predicate has been redefined by the node controller as `writeln(Term):-output(Term)` so this resulted in a series of messages of the following form to be sent back to the client, one per second:

```
{ "type": "output",
  "pid" : 56120932,
  "data": "hello" }
```

For this to work over HTTP, a trick somewhat similar to so-called *long-polling* is employed: every time a JSON structure of type `output` reaches the client, it must issue a new request for a response, which can come as more output, or some other response. See Section 3.6.2 for more details.

The call to `greetings/0` will run forever unless the user does something about it. When the user hits Control-C, the nonterminating query is aborted, and the node responds with the following message:

```
{ "type": "abort",
  "pid" : 56120932 }
```

If the client does not exit the remote process, the node may terminate it after a set time with no activity. The pid will no longer be valid and any changes made to the workspace of the process will be lost.

### 3.5.4 Browser talking to an ACTOR node

Unlike an ISOTOPE node, an ACTOR node gives clients access to the full set of Erlang-style concurrency predicates and exposes them over a bidirectional WebSocket connection. This combination enables push-driven interaction, mixed-initiative communication, and concurrent actor programming – capabilities that the HTTP-based APIs cannot support. Figure 3.11 illustrates a scenario in which a user engages with an ACTOR node over such a connection.

After having loaded the shell and established a WebSocket connection with the node, the JavaScript process sent the following command to the node's controller, instructing it to spawn a toplevel actor running a session with the clause `q(X):-p(X)` loaded into the actor's private database:

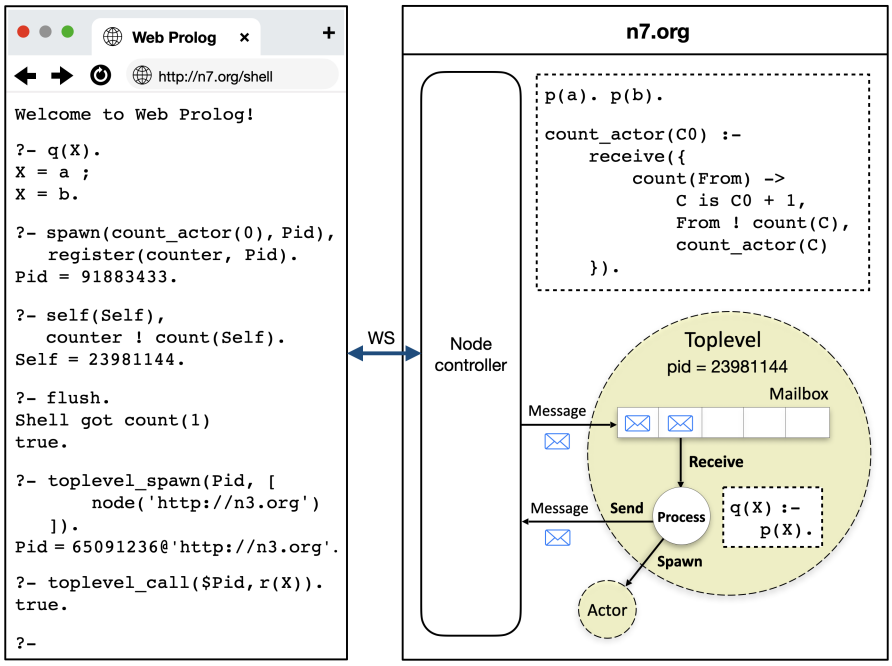



**Fig. 3.11** A scenario involving a web browser in conversation with a remote node running a toplevel actor over a stateful WebSocket connection.

```
{ "command": "toplevel_spawn",
  "options": "[session(true), load_text('q(X):-p(X).')]" }
```

As requested by the client, the pid of the toplevel actor was returned to the client, not in the form of a HTTP response, but embedded in a JSON formatted message sent over the WebSocket connection:

```
{ "type": "spawned",
  "pid" : 23981144 }
```

From here on, the solving of the query `?-q(X)` proceeds in a way very similar to how it worked in the ISOTOPE scenario, except that instead of the JavaScript shell making an HTTP GET request, it sent the following command message to the node over the WebSocket connection:

```
{ "command": "toplevel_call",
  "pid": 23981144,
  "goal": "q(X)",
  "options": "[limit(1)]" }
```

Knowing that the node's shared database offers a definition of `count_actor/1`, the user then decided to spawn a count server of the kind we saw in Chapter 2. Here is the message sent over the WebSocket connection:

```
{ "command": "toplevel_call",
  "pid": 23981144,
  "goal": "spawn(count_actor(0),Pid),register(counter,Pid)",
  "options": "[limit(1)]" }
```

This resulted in:

```
{ "type": "success",
  "pid" : 23981144,
  "data": [{"Pid":91883433}],
  "more": false }
```

While the toplevel actor continued to serve shell queries, the spawned count actor ran concurrently and received `count` messages independently of the shell interaction, illustrating the concurrent, actor based nature of the ACTOR profile. With access to the pid of the spawned count server, the user could send it a `count` message in order to try it out, and call `flush/0` to inspect the response.

Finally, the user called `toplevel_spawn/2` with the `node` option. This is yet another way to do network-transparent concurrency that will be dealt with in more detail in Chapter 4.

In this implementation, if the user leaves the terminal at `http://n7.org/shell`, closing the WebSocket causes the controller to terminate the session toplevel (via `exit/2`); before exiting, the toplevel terminates the spawned count actor. This yields a supervision-like hierarchy, discussed further in Chapter 4.

### 3.5.5 Social robot talking to an ACTOR node

In the previous scenarios, the shell and its toplevel actor served as the root of control: it was the first long-lived process from which other actors were spawned, monitored, and addressed. In general, however, the root of an application need not be a toplevel at all. Any actor can play that role, and a client may interact with such an actor directly, without routing interaction through a shell-style PTCP conversation.

Figure 3.12 sketches a deliberately minimal toy scenario in which a social robot interacts with a remote ACTOR node over the stateful WebSocket API.
The robot itself provides speech recognition and speech synthesis: it turns the user's spoken input into text, sends that text to the node, receives a reply as text, and speaks it back to the user. In this toy setup the remote side does nothing clever – it simply echoes the input – so the interaction is a "parrot" loop. The purpose is not to propose a useful robot skill, but to isolate the transport-level point: unlike the shell scenarios, the client is not a terminal driving a toplevel, and yet it can still engage in persistent, bidirectional interaction with a remote node.

At the protocol level, the scenario begins with the robot requesting that the node spawn an echo actor. The request is sent as a WebSocket message to the node controller (encoded here in JSON for the benefit of a non-Prolog client), and the

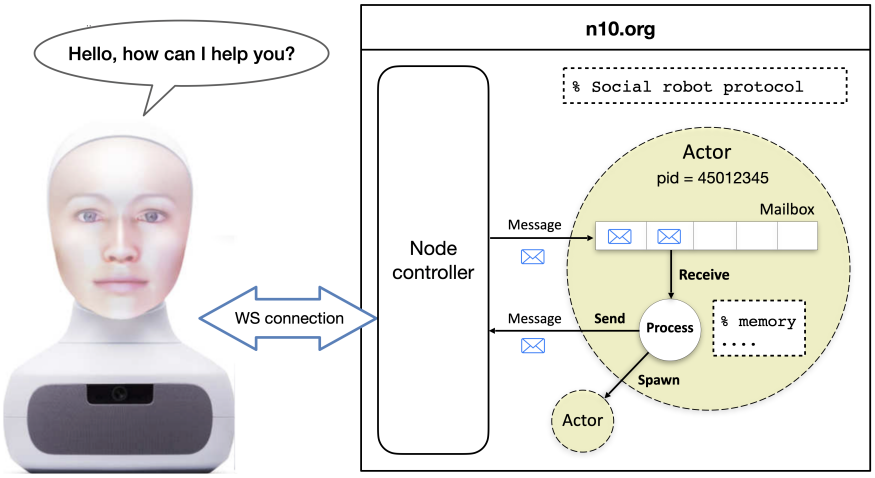



**Fig. 3.12** Stateful interaction in a scenario involving a social robot and a remote ACTOR node.

node responds by returning the pid of the spawned actor. This pid becomes the capability by which the robot can address the actor in subsequent messages.

```
{ "command": "spawn",
  "goal": "echo_actor",
  "options": "[monitor(true)]" }
```

```
{ "type": "spawned",
  "pid": 47110023 }
```

The WebSocket transport matters here because the interaction is push-driven and continuous. The robot establishes one connection and then exchanges messages in both directions as events occur: it sends new utterances as soon as they are recognised, and the node can return responses as soon as they are produced, without requiring a new HTTP request per response. This is the sense in which the interaction is stateful at the connection level: the channel remains open, latency is low, and messages can be delivered whenever either party chooses to speak.

In realistic conversational systems, of course, echoing recognised text is not enough. One typically needs additional components such as natural language understanding, a dialogue manager, grounding in a knowledge base, and response generation, as well as timing-sensitive coordination of speech, gaze, and facial expressions. We return to such agent-level concerns in Chapter 8; the present scenario only serves to show that, with the ACTOR profile, the transport and interaction model already supports the kind of persistent, low-latency, bidirectional exchange that embodied clients require.

### 3.5.6 Could there be other profiles?

The hierarchy RELATION ⊂ ISOBASE ⊂ ISOTOPE ⊂ ACTOR is useful because many applications need less than a full ACTOR node. Simpler profiles are easier to implement, govern, and deploy, while still supporting portable client behaviour.

At the same time, the hierarchy is not meant to be a closed taxonomy. The existing profiles mostly specify *what* capabilities are exposed at the node interface. One can also imagine profiles that express *how* a node chooses to do it: restrictions, guarantees, or disciplines that cut across the capability levels.

A PURE profile is one example: clients would be restricted to a declarative subset (excluding cut, non-logical arithmetic, and related extra-logical constructs), while still potentially allowing features such as source injection. Another example is a statechart discipline exposed as an optional ACTOR sub-profile for reactive conversational control. Whether such orthogonal profiles are worth standardising remains open, but the architecture allows them.

## 3.6 Notes on node implementation

This chapter specifies node behaviour primarily in terms of externally observable contracts: profiles describe what a client may rely upon, and the web APIs describe the shapes of requests and responses. Implementation techniques are not part of the contract, and different Prolog systems – and different host platforms – will naturally realise the contracts in different ways. Nevertheless, a few recurring engineering issues are worth highlighting. They explain several design choices, clarify why multiple APIs are worth supporting, and help readers anticipate the practical costs associated with each profile.

### 3.6.1 Statelessness as a protocol property

The stateless HTTP API is *stateless for the client*: each request contains all information needed to interpret it, and the node is not required to remember a conversational session in order to respond. This does not prohibit the node from keeping internal caches. It merely means that caches are best-effort optimisations whose presence must not be observable as a behavioural dependency. A node may evict cached state at any time and still remain correct and conformant.

As noted in Section 3.5.1, statelessness brings familiar web virtues: meaningful URIs, replayable requests, and cacheable responses. A corresponding limitation is that a stateless request cannot be *externally* interrupted by the client once evaluation has begun. There is no control channel associated with the computation, and no session actor to signal. In practice, therefore, the stateless API must rely on

node-enforced resource policy, most importantly running timeouts, to avoid nonterminating or overly expensive computations.

#### Two strategies for implementing paged solution retrieval

The `offset` and `limit` parameters make it possible to retrieve solutions in slices. This subsection is concerned only with how a node might implement that contract efficiently; the architectural interpretation of paged retrieval as an instance of Prolog-style backtracking is discussed separately in Chapter 4.

A naive implementation uses *restart-and-skip*: to compute the slice starting at offset $N$, the node restarts evaluation from the beginning, discards the first $N$ solutions, and then collects up to `limit` further solutions. This strategy is simple and portable, but it may waste work when later slices are requested, because earlier solutions are recomputed only to be thrown away.

A more efficient strategy uses *save-and-resume*. The node evaluates the goal in an execution context that can be suspended after producing a slice and resumed later to continue search. The node may maintain a cache keyed by a fingerprint of the query (goal, template, and any relevant execution parameters) together with the next slice boundary. When a request arrives for offset $N$, the node first attempts to resume a cached computation that has already produced $N$ solutions; if none is available, it falls back to restart-and-skip. This improves performance in the common case without changing the protocol contract.

Importantly, the choice between these strategies is not visible at the API level. Clients only observe that slices are returned in Prolog order and that the `more` flag correctly reports whether further solutions may exist. All optimisation is therefore constrained by the requirement that cached state is disposable: eviction must never change the meaning of subsequent responses, only their cost.

### 3.6.2 Semi-stateful HTTP sessions and response retrieval

The semi-stateful HTTP API differs from the stateless API in one crucial respect: each request targets a long-lived toplevel actor, identified by a pid, and the observable behaviour of the conversation depends on the evolving state of that actor. Nevertheless, each individual HTTP exchange is still an ordinary request–response transaction. In this sense, the protocol is *semi-stateful*: the transport remains stateless, but the addressed computation is not.

From an implementation perspective, this style aligns naturally with the architecture of SWI-Prolog's `library(pengines)` and systems such as SWISH: the server maintains per-session engines, while the client repeatedly issues short requests that either advance the computation, or retrieve the next available response produced by that computation.

Output and prompts fit this model once they are treated as ordinary protocol events emitted by the session actor, rather than as exceptional side effects. Concretely, the toplevel actor can be understood as producing a sequence of *responses* (answers, failure, errors, output fragments, prompts, and acknowledgements) that the client retrieves over a series of short HTTP exchanges.

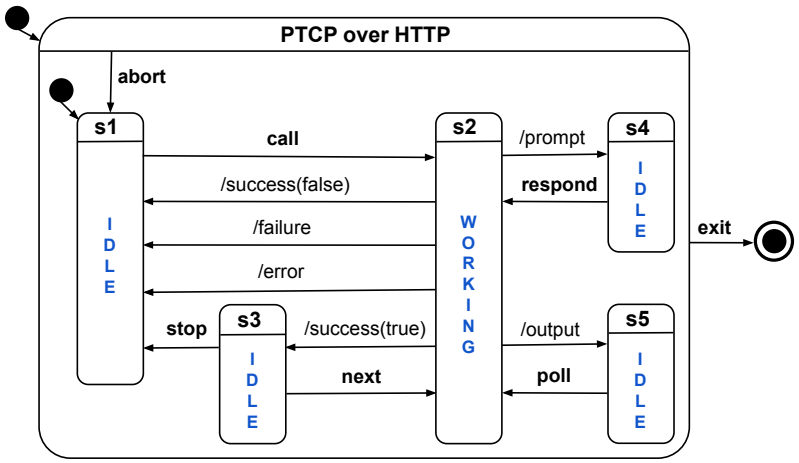



**Fig. 3.13** Statechart specifying the PTCP for a successful conversation with a toplevel over HTTP. The transitions are labeled with *message types*. Types in bold are sent from the client to the toplevel, whereas message types with a leading / goes in the opposite direction, from the toplevel to the client.

The statechart in Figure 3.13 extends the core PTCP with the states and transitions needed for HTTP delivery. When the toplevel emits an `/output` event during evaluation, the client enters a local pull cycle: it retrieves the pending event and immediately issues a follow-up **poll** request for the next pending response. This cycle repeats until the toplevel produces a terminal response (`/success`, `/failure`, `/error`), at which point control returns to the main protocol states. The mechanism resembles long-polling only locally – within the scope of a single goal evaluation that happens to produce intermediate events – rather than as a general-purpose polling loop.

Prompts require one additional step: when the next pending response is a prompt, the client retrieves it, presents it to the user, and sends a follow-up request that delivers the user's reply back to the session actor. From the protocol's perspective, this is still a request–response rhythm and no polling is needed here; the client is simply advancing the conversation by alternately retrieving pending events and, when prompted, supplying input. Ordering is preserved because responses are dequeued in the order they were produced. Transient transport failures may delay retrieval but do not change the logical meaning of the session.

### 3.6.3 Stateful WebSockets and push-driven interaction

The stateful WebSocket API exists for interactions that are naturally push-driven. Once a connection is established, the node can deliver answers, output fragments, prompts, and monitor events as soon as they are produced, without requiring a new client request for each response. This improves latency and avoids the request choreography of our long-polling trick. It also supports interleaved traffic from multiple actors over one connection, which is central to ACTOR-profile use cases.

Implementation complexity, however, shifts from request handling to connection lifecycle management. A robust node must maintain per-connection routing state (which pids belong to which client), preserve ordering within each actor stream, and enforce outbound back-pressure so that a slow client cannot cause unbounded buffering on the node. It must also define clear disconnect semantics: when a client vanishes, session actors and child actors are either terminated or detached according to policy, and monitored parties observe the corresponding `down`/`exit` events. In this sense, the WebSocket API is conceptually simple but operationally demanding: the protocol surface is small, while lifecycle and resource handling must be precise.

## 3.7 Summary

Chapter 2 equipped Web Prolog with actors and concurrency, and this chapter lifted those ideas to the level of Prolog agents and nodes. The central move was to treat toplevels as stateful actors governed by an explicit Prolog Toplevel Communication Protocol (PTCP). By making the conversational contract first-class – including how queries are started, how solutions are streamed, how I/O and aborts are handled, and how sessions are terminated – we turned an informal REPL interaction pattern into a well-defined behavior that can be implemented inside a node or exposed over the network.

On top of this, we introduced Prolog nodes as agents in their own right. A node is more than a mere container for code and data: it offers one or more toplevels, may host arbitrary actors, and is addressable on the Prolog Web through URIs and web APIs. The notion of node profiles (RELATION, ISOBASE, ISOTOPE, ACTOR) makes explicit which capabilities a node must offer to its clients. In particular, the ACTOR profile marks nodes that support the full actor model and PTCP-based toplevels, whereas the more restricted profiles support progressively smaller subsets of those capabilities. This lets implementors scale their commitment, and lets clients reason about what they can rely upon in a heterogeneous network.

A recurring theme in the chapter was the distinction between private state and shared data. Actors and toplevels maintain their own conversational and internal state – their "private beliefs" – while nodes may expose shared knowledge bases that serve as common background for all actors running on that node. The dynamic database, though available within an actor, remains process-local and cannot be used for inter-

process communication, preserving the principle that processes should communicate to share memory rather than share memory to communicate.

The three communication paradigms – stateless HTTP, semi-stateful HTTP, and stateful WebSocket – serve complementary purposes. Stateless interactions offer simplicity, cacheability, and horizontal scalability; semi-stateful sessions enable dynamic database updates without persistent connections; stateful WebSockets unlock the full power of bidirectional, real-time interaction and concurrent actor programming. Supporting all three within a single node architecture allows applications to choose the appropriate trade-off between simplicity and capability.

Finally, the chapter touched on security considerations. Like JavaScript, Web Prolog is designed as a sandboxed language: it omits file I/O, socket programming, and direct OS access, relying instead on the host environment for such capabilities. Security in the actor model rests primarily on name distribution – if a process identifier is never revealed, the process cannot be accessed. This principle, combined with the constrained language subset and profile-based capability advertisement, forms the foundation of a security model appropriate for running untrusted code on shared infrastructure.

The overall message of the chapter is that Web Prolog is not just a language but also a way of packaging processes, protocols, and capabilities into networked agents that can be composed and reasoned about at a higher level of abstraction. Chapter 4 extends these local concepts across the network, showing how nodes discover each other, how actors migrate and communicate across node boundaries, and how the Prolog Web emerges as a federated infrastructure for distributed logic programming.

The motivation for profiles is not only security, but also coordination. FYPB emphasises that portability is not just about language constructs, but also about portable libraries and developer tools – for unit testing, documentation, linting, and package distribution. A profile is a way to make that expectation concrete: it specifies which predicates, libraries, and APIs are guaranteed to exist on a node of a given kind. In other words, profiles separate *observable capability* from *implementation commitment*, enabling tools to target stable capability sets even when different implementations continue to innovate internally.

# Chapter 4
# The Prolog Web

> Imagine the Web wrapped in Prolog, running on top of a distributed architecture – a network of nodes exposing HTTP and WebSocket APIs, speaking web formats such as JSON. Think of it as a high-level Web, with Prolog agents capable of serving answers to queries – answers that follow from what the Web knows. Moreover, imagine it being programmable, allowing Web Prolog source code to flow in either direction: from client to node, or from node to client. This is what **the Prolog Web** is all about.
>
> *The Prolog Web – the elevator pitch*

The previous chapters introduced Web Prolog as a language and Prolog agents as programmable entities. This chapter turns to the third element of the Prolog Trinity: the *Prolog Web* itself – a network of nodes hosting actors that communicate across node boundaries. In short, we extend Web Prolog into a language for distributed programming.

The model draws on Erlang's *network-transparent concurrency*: if we know a node's identity and we are authorised, we can spawn an actor there; if we know an actor's pid or its registered name, we can send it a message, even across node boundaries. Unlike an Erlang cluster, however, the Prolog Web is as open as the Web itself. Here, *openness* means more than mere reachability. It means heterogeneous ownership, weak trust assumptions, communication over ordinary Web infrastructure, and an environment in which latency, disconnects, and partial failure are part of normal operation. Openness therefore brings both opportunity and risk, so the chapter pays close attention to safety: how lifetime discipline and capability separation prevent orphan processes and resource leaks.

Openness also forces a basic separation of authority. In the Prolog Web there are two roles: *clients*, who temporarily borrow execution resources in order to run queries and coordinate short-lived concurrent work, and *node owners*, who are responsible for the node as an ongoing public resource. For clients, the default discipline is *containment-by-default*: spawned processes remain tied to the client session and are torn down when the session ends, preventing orphans and resource leaks. For owners, the corresponding power is *provisioning*: the ability to install node-resident predicates and start long-lived actor services meant to be shared and kept available.

In practice such services are often managed under supervision, but the generic supervision patterns that support this belong to Chapter 5. This role split will recur in concrete form when we discuss remote placement and the various code-shipping mechanisms via the `load_*` predicates.

The chapter is organised in five sections, moving from the concurrent foundation through progressively higher layers of abstraction. The concurrent Prolog Web (Section 4.1) extends the actor model of Chapters 2 and 3 across node boundaries: remote spawning, cross-node messaging, orphan prevention, and the unified semantics that treats local and remote communication as instances of the same model. With the concurrent substrate in place, we can already speak concretely about node-resident services: long-lived actors started by the node owner, published under stable names, and invoked by clients over the Prolog Web. The reusable engineering patterns that structure such services – servers, supervisors, and other generic behaviours – are introduced in the next chapter and implemented in Chapter 5.

The sequential Prolog Web (Section 4.2) then introduces a synchronous, nondeterministic abstraction built on top of the concurrent layer. The central construct, `rpc/2-3`, can be understood as a disciplined use of remote toplevel actors, though in practice it is implemented over stateless HTTP backed by cached toplevels operating in save-and-resume mode. The dependency is genuine: the sequential interface hides the actor machinery, but it is always, at bottom, built from it.

The programmable Prolog Web (Section 4.3) shows how code and data flow between client and node: clients may inject task-specific predicates into their own execution contexts, while node owners install predicates and service actors that define what the node offers.

A discussion of timing and failure boundaries (Section 4.4) makes explicit where waiting becomes observable and how timeouts attach to different interaction models. Finally, a section on security (Section 4.5) asks what makes it safe to let strangers run code on a node, and answers in terms of three structural mechanisms – inexpressibility, invisibility, and containment – together with the deployment concerns that complete the picture. Chapter 9 later returns to the architectural and paradigmatic questions that these technical sections raise.

## 4.1 The concurrent Prolog Web

As introduced above, all the mechanisms described in the preceding chapters – spawning, sending, linking, monitoring, registering – work transparently across node boundaries. This section makes that claim concrete by introducing remote placement via the `node` option and showing how actor-style interaction spans nodes. The focus is on *where* an actor runs and how it exchanges messages across the network, but also on the safety and authority questions that network-transparent concurrency raises on an open Web. Programmability mechanisms such as `load_*` options are deliberately postponed to Section 4.3.

### 4.1.1 Actor talking to remote actor

For specifying the node on which we wish to create an actor process, `spawn/3` can be passed the `node` option:

```
?- spawn(echo_actor, Pid, [
       node('http://n8.org'),
       monitor(true)
   ]),
   register(echo_actor, Pid).
Pid = 99231321@'http://n8.org'.
?-
```

Note that the pid is a compound term: an integer paired with a URI using the infix operator `@`. If `n8.org` is a node conforming to the ACTOR profile, and if the node's shared database defines the predicate `echo_actor/0`, then we can have a conversation with our remote echo server like so:

```
?- self(Self),
   echo_actor ! echo(Self, hello).
Self = 71773120.
?- flush.
Shell got echo(hello)
true.
?- whereis(echo_actor, Pid),
   exit(Pid, die).
Pid = 99231321@'http://n8.org'.
?- receive({A -> true}).
A = down(99231321@'http://n8.org',99231321@'http://n8.org',die).
?-
```

### 4.1.2 Actors playing ping-pong while on different nodes

The actors playing ping-pong back in Chapter 2 kept their game inside one node. Passing the option `node('http://n2.org')` to one of the calls to `spawn/3` allows the game to be played between two nodes:

```
ping_pong :-
    spawn(pong, Pong_Pid, [
        node('http://n2.org')
    ]),
    spawn(ping(3, Pong_Pid)).
```

Figure 4.1 illustrates this scenario.

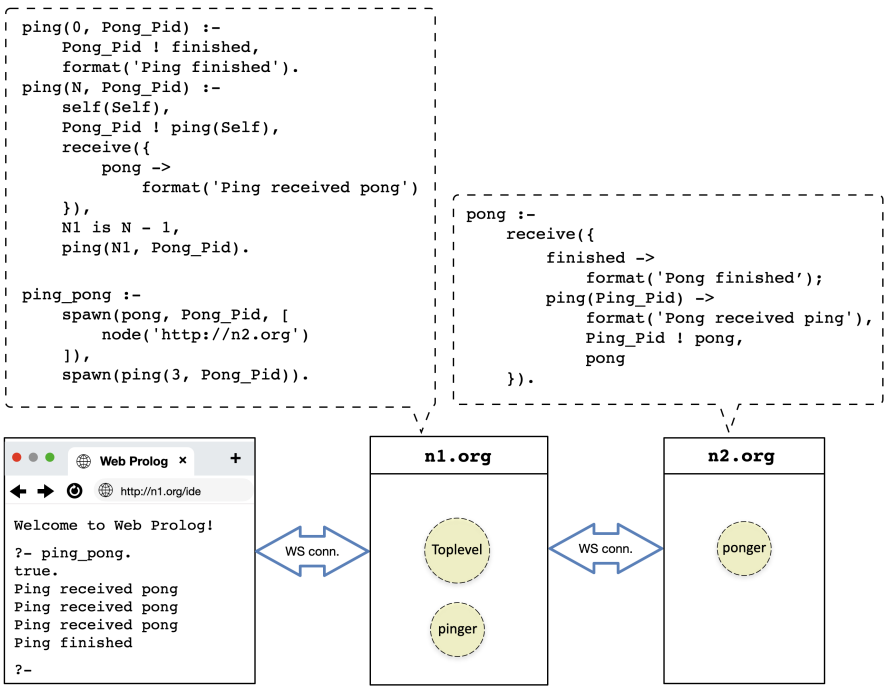



**Fig. 4.1** Actors playing ping-pong while on different nodes.

But why do we get the output written by the "pinger" only, and not the output from the "ponger"? The asymmetry in the output is not an accident of the example but a consequence of a deliberate capability rule. A terminal is attached to one particular toplevel actor, and rendering output in that terminal is treated as an *I/O capability* that is scoped to the toplevel's local process lineage. Concretely, only actors that (i) run on the same node as the toplevel to which the terminal is attached, and (ii) are local descendants of that toplevel, may write output that is forwarded to the client for display. In any case, readers may rest assured that the actors really *do* play ping-pong.

### 4.1.3 The node option works for toplevels too!

Toplevels are actors, so it should come as no surprise that the `node` option can also be used to create a toplevel process on a remote node:

```
?- toplevel_spawn(Pid, [
       node('http://n1.org')
   ]).
Pid = 83110912@'http://n1.org'.
```

```
?-
```

The query `?-human(Who)` executes on the remote node `n1.org`, with answers returned to the local shell:

```
?- toplevel_call($Pid, human(Who), [
       template(Who),
       limit(1)
   ]).
true.
?- receive({Msg -> true}).
Msg = success(83110912@'http://n1.org',[plato],true).
?- toplevel_next($Pid).
true.
?- receive({Msg -> true}).
Msg = success(83110912@'http://n1.org',[aristotle],false).
?-
```

Network transparency makes distributed concurrency easy to *use*. The harder question is how to make it *safe* on an open, failure-prone Web: how do we prevent orphan processes, constrain resource usage, and ensure that failure leads to reclamation rather than silent accumulation? The answer is largely about lifetime discipline, supervision, and the separation of authority between clients and node owners.

### 4.1.4 Node-resident actor processes

In addition to predicates in the shared database, the owner of a node may install *node-resident actor processes* – long-lived actors that are intended to be durable, shared, and continuously available. They are usually of a kind that we do not want unauthorized external clients to be able to create or destroy; doing this is the privilege of the node owner. This is also where supervision trees become most useful: crash recovery for node-resident services is part of the node owner's responsibility, and Chapter 5 introduces the supervisor behaviour that makes this responsibility tractable.

#### A node-resident count actor

We begin with a deliberately small example. Assume that the owner of the node `n1.org` starts a count actor and gives it a stable registered name:

```
?- spawn(count_actor(0), Pid),
   register(counter, Pid).
```

The registered name can now be used instead of the pid when sending messages to the process:

```
?- self(Self).
Self = 51230945.
?- counter@'http://n1.org' ! count($Self),
   receive({Count -> true}).
Count = 1.
?- counter@'http://n1.org' ! count($Self),
   receive({Count -> true}).
Count = 3.
?-
```

The example is intentionally simple, but it already shows an important shift in perspective. This actor is not a client-local helper process created for one temporary task; it is a node-resident process with a stable public name, and it therefore behaves like a shared stateful resource. The fact that the second reply is `3` rather than `2` is not an accident of presentation, but evidence that the actor's state is shared across interactions: another client has incremented the counter in the meantime.

In other words, once a node owner chooses to run and name an actor in this way, the actor begins to look less like a private process and more like a service. The next subsection develops this idea further by showing a publish-subscribe actor whose service character is even more explicit.

### A node-resident publish-subscribe service

The same service perspective becomes even clearer in a publish-subscribe example. Consider the following actor:

```
pubsub_actor(Subscribers0) :-
    receive({
        publish(Msg) ->
            forall(member(Pid, Subscribers0), Pid ! msg(Msg)),
            pubsub_actor(Subscribers0);
        subscribe(Pid) ->
            pubsub_actor([Pid|Subscribers0]);
        unsubscribe(Pid) ->
            (   select(Pid, Subscribers0, Subscribers)
            ->  pubsub_actor(Subscribers)
            ;   pubsub_actor(Subscribers0)
            )
    }).
```

This actor maintains a list of subscribers. A `subscribe(Pid)` message adds a subscriber, an `unsubscribe(Pid)` message removes one, and a `publish(Msg)` message broadcasts `Msg` to all currently subscribed processes. On its own, this is just another actor definition. It becomes a *service* when the node owner chooses to start it as a durable node-resident process and publish it under a stable name:

```
?- spawn(pubsub_actor([]), Pid),
   register(pubsub_service, Pid).
```

A client can then subscribe to the service and wait for messages from it:

```
?- self(Self),
   pubsub_service@'http://n8.org' ! subscribe(Self),
   repeat,
   writeln("Waiting for a message ..."),
   receive({
       msg(Message) ->
           format("Received: ~p~n", [Message]),
           fail
   }).
Waiting for a message ...
Received: hello
Waiting for a message ...
```

The message `hello` was received because some client published it to the same named actor:

```
?- pubsub_service@'http://n8.org' ! publish(hello).
true.
?-
```

The important point is not the specific protocol, but the change in status. Once the actor is started by the node owner, registered under a stable name, and intended for repeated use by multiple independent clients, it should be understood as part of the node's public surface. It is no longer merely a process that happens to be running; it is a shared conversational resource offered by the node.

This example also illustrates a useful contrast with the counter actor. The counter service exposes shared *state*: what one client observes depends on what other clients have already done. The publish-subscribe service instead exposes shared *coordination*: clients interact indirectly through a common mediating process. In both cases, however, the essential pattern is the same. A node-resident actor with a stable public name behaves as a service.

The implementation shown here is intentionally minimal. In practice, one would typically place such a service under supervision, possibly persist subscriptions in a more durable store, and perhaps attach access control or topic structure. Those are engineering refinements. The conceptual point is already visible in the small example: named node-resident actors provide a natural way to realise stateful services on the Prolog Web.

### Naming, discovery, and service publication

The preceding examples rely on stable service names – `counter`, `pubsub_service` – to make node-resident actors reachable by clients. In this design, these are not ordi-

nary client-managed registrations. A client may still use `register/2`, `whereis/2`, and `unregister/1` for actors that it has itself created and named, but published services live in a separate owner-controlled service registry. Clients may send to a published service name, but they do not thereby gain the ability to discover the service's pid or to remove its published name. This raises the practical question: how do clients discover which services a node makes available?

The standard design principle is to separate a stable, location-independent identifier from the service's current location (van Steen and Tanenbaum, 2023). The Prolog Web inherits this directly: a service name should not bind clients to a particular pid, host, transport endpoint, or deployment topology. Services may be restarted, moved, replicated, or versioned; the published name remains stable.

This also clarifies the distinction between ordinary naming and service publication. Ordinary registration is a convenience for private or client-local actors. Service publication is a different act: the node owner places a name into the node's public service namespace. An implementation should therefore provide dedicated operations – `register_service/2` and `unregister_service/1` – for published services, leaving the ordinary naming predicates available for client-managed actors.

The safe pattern is to treat discovery as an owner-curated *publication mechanism*: the node exposes a directory that lists only those services intended as part of its public surface. Discovery should resolve *published* names, not reveal the existence of everything that happens to be running.

This is a design invariant, not merely a preference. If clients could enumerate all running processes, or if ordinary client naming operations exposed or removed published services, discovery would collapse into introspection and name discipline would be lost. The security implications of this principle are developed in Section 4.5.2.

In the Trinity, the cleanest way to expose such a directory is to make it queryable through `rpc/2-3`. Concretely, a node that wishes to advertise services may provide a relation such as `service/2` (or `service/3`) in its shared database, for example:

```
service(counter, meta(actor, protocol(count_v1))).
service(pubsub_service, meta(actor, protocol(pubsub_v1))).
```

A client can then discover advertised services using an ordinary goal-oriented query:

```
?- rpc('http://n1.org', service(Name, Meta)).
Name = counter,
Meta = meta(actor, protocol(count_v1)) ;
Name = pubsub_service,
Meta = meta(actor, protocol(pubsub_v1)).
?-
```

This keeps the design uniform. Service discovery remains declarative and compositional: clients can ask for all services, filter by protocol, restrict to public entries, or combine discovery with other constraints in the same query. At the same time, it does not weaken the security model, because the node remains free to decide what

it advertises, to require authorisation for the directory predicate, and to ensure that the directory reveals only what it has chosen to publish.

With this distinction in hand, the earlier counter example can be read more precisely. Once an actor is published under a stable service name such as `counter`, it is no longer merely a running process known to its owner, but part of the node's public surface. In that role it behaves as a shared service: multiple clients may interact with it through the same published name, while the underlying pid remains an implementation detail of the node. Observations such as receiving `3` rather than `2` are then best understood not as oddities, but as visible consequences of shared state.

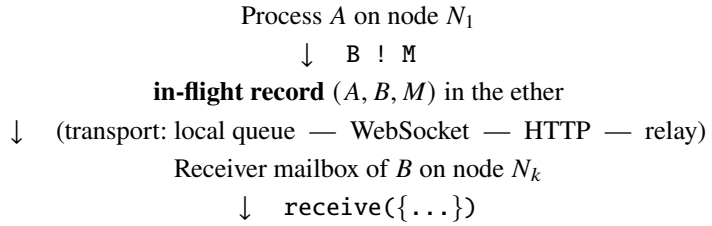
### 4.1.5 Uniform messaging across node boundaries

The preceding examples show that the same primitives – spawning, sending, linking, monitoring, registering – work whether the target process is local or remote. This uniformity is not accidental; it rests on a deliberate design choice inherited from the unified semantics proposed by Svensson et al. (2010) for Erlang-like systems. The key idea is to eliminate the semantic distinction between local and remote communication: all side-effecting operations are mediated by a node controller abstraction, so that placement becomes a configuration detail rather than a semantic one. Chapter 9 returns to the deeper implications of this distribution-first stance; here we spell out the messaging model that makes it concrete.

#### Two-step messaging model

Following Svensson et al. (Svensson et al., 2010)'s *ether* abstraction, we model messaging as a two-step mechanism: *sending* places an in-flight record into a virtual system queue (the *ether*), and a later *delivery* step appends the message to the receiver's mailbox. This separation allows us to define program meaning independently of transport.

The in-flight layer is a semantic device rather than an implementation requirement. Locally, a runtime may realize sending by inserting the message directly into the receiver's mailbox, which can be seen as choosing a schedule in which each send is immediately followed by its delivery. In Web Prolog, the in-flight phase may be implemented using multiple transports, as illustrated below.



The transport realizations in Web Prolog are as follows.

- **Local delivery**: the runtime enqueues $(A, B, M)$ in an internal scheduler structure and later appends $M$ to $B$'s mailbox.
- **Direct remote delivery**: $(A, B, M)$ is serialized and shipped from $N_1$ to $N_k$ as a WebSocket frame and then enqueued for mailbox delivery.
- **Request-based delivery**: $(A, B, M)$ is carried in an HTTP request to $N_k$ (or to a node gateway), which authenticates, deserializes, and enqueues the message.
- **Relay-mediated delivery**: $(A, B, M)$ is sent to an intermediary that forwards it to $N_k$, after which normal mailbox delivery occurs.

This separation lets us specify one uniform concurrency model while allowing multiple deployment choices. Ordering guarantees are stated over *delivery* into mailboxes: per sender-receiver FIFO when sender and receiver are addressed without mixing addressing modes, with no global order across different senders (Section **??**). Liveness properties depend on the fairness of the delivery schedule rather than on the specifics of the network path.

## 4.2 The sequential Prolog Web

The *sequential Prolog Web* is the fragment of the Prolog Web in which every client interaction can be explained by ordinary Prolog-style backtracking. More precisely, a node belongs to the sequential Prolog Web if, for every client session, the sequence of answer substitutions returned for each goal is exactly the sequence that a single-threaded SLD-style execution would produce. No reasoning about interleavings, mailboxes, or concurrent actors is needed to understand what happens – the entire observable behaviour admits explanation by a single resumable logical history.

The *concurrent* partition is fundamentally different: it admits multiple simultaneous causal histories, requires reasoning about interleavings, and cannot in general be reduced to a single sequential trace. Yet the two partitions are not independent. The sequential layer of the Prolog Web is built directly on top of the concurrent one, and understanding that dependency is the key to understanding why the sequential interface looks and behaves as it does.

The full layering, from bottom to top, is as follows:

1. Actor processes and asynchronous messagin.
2. Toplevel actors, which add goal evaluation and answer-stream management to the basic actor model.
3. `rpc/2-3` over a remote toplevel – the conceptual implementation that makes the synchronous, nondeterministic interface explicit.
4. `rpc/2-3` over stateless HTTP – the practical implementation, backed by cached toplevels operating in save-and-resume mode, which realizes the same interface more efficiently.

Items 3 and 4 expose the same interface; they differ in realization, not in semantics. Each step in the stack thins the interface and hides more of the underlying machinery.

Because the concurrent layer has already been established, we can now present `rpc/2-3` and its companions `promise/3-4` and `yield/2-3` with the reader already understanding what lies beneath. Not every distributed program needs explicit concurrency – often a caller simply wants to ask another node to solve a goal and wait for answers – but the sequential interface is always, at bottom, a disciplined use of the concurrent one.

### 4.2.1 Nondeterministic remote procedure calls

In Web Prolog, `rpc/2-3` is a high-level meta-predicate for making *nondeterministic remote procedure calls*. It allows a process running on a node $N_1$ to call and attempt to solve a goal in the Prolog context of another node $N_2$, taking advantage of the programs and data offered by $N_2$ as if they were local to $N_1$. Such calls are synchronous: the caller blocks while waiting for answers. In return, `rpc/2-3` is remarkably easy to use for distributing programs across two or more nodes. It can therefore be seen as a construct for distributed programming that trades concurrency for ease of programming.

A Web Prolog client performs a remote call by providing a URI identifying the node and a goal to be solved in that node's context. The optional third argument is a list of options. Here is a simple example:

```
?- rpc('http://n1.org', human(Who)).
Who = plato ;
Who = aristotle.
?-
```

Note that solutions are returned on demand and in the familiar one-solution-at-a-time fashion: variable bindings appear as instantiations of variables occurring in the goal.

#### Using rpc/3 with the limit option

The `rpc/3` predicate supports a `limit` option, inherited from the underlying `toplevel_call/3` predicate. Passing `limit(K)` instructs the remote node to compute at most $K$ solutions per network roundtrip. The default is `infinite`, meaning all solutions are fetched in a single exchange. Here is what happens if we override that default with `limit(1)`:

```
?- rpc('http://n1.org', human(Who), [
       limit(1)
   ]).
Who = plato ;
Who = aristotle.
```

```
?-
```

From the caller's perspective, the result is identical. What changed is invisible: instead of retrieving both solutions in a single roundtrip, the call now required *two* – one per solution. In this case, the option adds cost without benefit. Later, we shall see contexts in which limiting the number of solutions per roundtrip matters in a genuinely positive way.

In addition to the `limit` option, `rpc/3` inherits the `load_*` options from `spawn/3` and the `timeout` and `once` options from the toplevel mechanism. Examples using `load_*` options will be given in Section 4.3.1. See Appendix A for details about the other options.

### The semantics of rpc/2-3

For intuition it is useful to note that a call to `rpc(URI,Goal)` is semantically equivalent to the following pattern, which makes explicit that solutions arrive as instantiations of a copy that is then unified back into the original goal:

```
copy_term(Goal, Copy),
call(Copy),           % executed on node at URI
Goal = Copy.
```

The key implication is that no variables are shared across nodes: the remote execution operates on a renamed-apart copy, and only the induced substitution is returned to the caller via unification with the original goal.

This three-line characterisation is intended as a *logical* specification of `rpc/2-3`: it defines what answers the caller may observe, not how those answers are obtained. As a specification it is precise and complete for the intended domain. Web Prolog restricts remote goals to pure Herbrand terms – attributed variables are not supported, and the serialisation protocol ensures that bindings returned to the caller are always meaningful in the caller's context, ruling out node-local references such as stream handles or database pointers. What the characterisation deliberately abstracts over is the operational machinery required to deliver those answers across a network: the creation of a remote process, the protocol that sustains nondeterminism across multiple solutions, and the handling of timeouts, exceptions, and partial failure. These concerns are addressed by actor infrastructure described in the sections that follow.

A notable property of `rpc/2-3` is that it preserves the logical purity of the goal it calls: if the goal is pure, then the answers returned by `rpc/2-3` are exactly the logical consequences of the remote database – the network introduces no additional bindings, side effects, or observable state changes.[1] This holds at the level of answer substitutions, assuming successful communication; in practice, timeouts, connection failures, and resource limits may prevent all answers from being delivered, but they

[1] This is a property it shares with the `call/N` family of Prolog predicates.

do not corrupt the answers that are delivered. Later, we return to this observation when discussing the idea of a *pure* Prolog Web.

### 4.2.2 Implementing `rpc/2-3` on top of a toplevel actor

The central construct of the sequential layer, `rpc/2-3`, can be implemented straightforwardly on top of a toplevel actor: spawn a toplevel on the target node, submit a goal, and collect answer messages in a loop. From the caller's perspective the interaction is synchronous and nondeterministic – variables are bound on success, backtracking yields the next solution – but underneath, the familiar PTCP communication protocol is doing the work:

```
rpc(URI, Goal) :-
    rpc(URI, Goal, []).

rpc(URI, Goal, Options) :-
    toplevel_spawn(Pid, [
         node(URI),
         session(false),
         monitor(false),
       | Options
    ]),
    toplevel_call(Pid, Goal, Options),
    wait_answer(Pid, Goal).
```

The remote toplevel is spawned with `session(false)` so that it terminates after the query has been exhausted. The implementation does not monitor the spawned process, so no `down` message needs to be handled. The loop can therefore terminate cleanly on failure, error, or after the last slice has been consumed:

```
wait_answer(Pid, Goal) :-
    receive({
        success(Pid, Slice, true) ->
            (   member(Goal, Slice)
            ;   toplevel_next(Pid),
                wait_answer(Pid, Goal)
            ) ;
        success(Pid, Slice, false) ->
            member(Goal, Slice) ;
        failure(Pid) -> !, fail ;
        error(Pid, Error) ->
            throw(Error)
    }).
```

The nondeterministic behaviour arises from two sources. An answer term of the form `success(Pid, Slice, ...)` message contains a *slice* of solutions, and `member(Goal, Slice)` enumerates them without further network communication. Only when a slice is exhausted does execution fall through to the second disjunct, which requests a new slice via `toplevel_next/1`. In this way, deterministic message exchanges with a remote toplevel are turned into a nondeterministic, on-demand interface at the caller.

Since the caller and callee do not perform independent work during overlapping periods, this is a purely synchronous and sequential pattern: `rpc/2-3` contributes to the sequential fragment of the Prolog Web.

### 4.2.3 Implementing `rpc/2-3` on top of the stateless HTTP API

Looking at the implementation of `rpc/2-3` in terms of the lower-level `toplevel_*` predicates makes visible what the high-level specification deliberately abstracts over. In particular, it becomes clear that all communication follows a strict request-response alternation: the caller creates a query remotely, then repeatedly asks for the next answer, blocking each time until a response arrives. There are no unsolicited messages, no server-initiated notifications, no concurrent streams – just a disciplined exchange in which every message from the client is answered by exactly one message from the server. This is precisely the traffic pattern that a stateless HTTP API is designed to carry, and it is why `rpc/2-3` can be implemented entirely on top of the sequential layer's HTTP interface without any loss of expressiveness.

This implementation is not merely adequate but arguably preferable. Because each exchange is a self-contained HTTP request-response pair, there is less risk of leaving orphan queries behind on the remote node. The approach works between any two ISOBASE-conforming nodes regardless of whether they support the programmable layer. It passes cleanly through firewalls, proxies, and load balancers. Failures are unambiguous – every request either receives a response or times out. And the entire conversation is inspectable with standard HTTP tooling. In short, for the synchronous, one-answer-at-a-time pattern that `rpc/2-3` requires, stateless HTTP is not a compromise but the right fit.

Compared to the WebSocket-based implementation shown above, the HTTP version is more involved. The additional complexity lies not in the control structure, which is shared between the two, but in transport concerns: serialising goals and templates, encoding terms correctly, and handling authentication, redirects, TLS settings, and timeouts. The full SWI-Prolog implementation, including URL construction, term encoding, and HTTP option handling, is given in Appendix B.

### 4.2.4 Browser talking to a node talking to a node

In the scenario depicted in Figure 4.2, a user of a Prolog terminal running in a browser interacts with a remote node on the Prolog Web, which in turn interacts with another node.

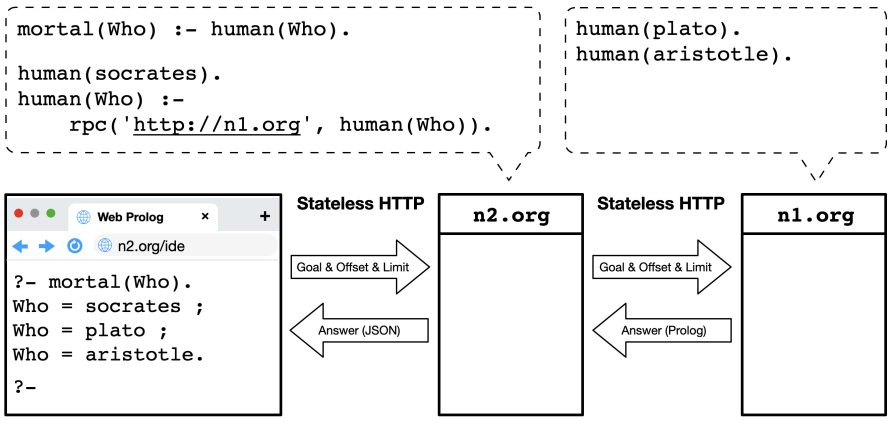



**Fig. 4.2** Stateless interaction in a scenario involving a web browser and two remote nodes.

The solutions shown in the terminal follow from the *combination* of two programs: the code on the upper left in Figure 4.2 belongs to `n2.org`, and it contains a URI pointing to the code offered by `n1.org`. The browser-to-node interaction as well as the node-to-node interaction take place over HTTP and are stateless in the sense that each request can be understood and processed without relying on server-side session state.

To retrieve the first solution to `?-mortal(Who)`, the browser issues a request to `n2.org` asking for exactly one solution:

```
GET http://n2.org/call?goal=mortal(Who)&limit=1
```

Since the user requests solutions one-by-one, the response contains a single binding together with a flag indicating that more solutions exist:

```
{ "type": "success",
  "data": [{"Who":"socrates"}],
  "more": true }
```

When the user types a semicolon, the browser requests the next solution. In this example, producing that next solution requires `n2.org` to consult `n1.org`. Concretely, `n2.org` evaluates `mortal/1` and, as part of doing so, triggers an `rpc/2-3` call for `human(Who)` against `n1.org`. Operationally, this involves an HTTP request to `n1.org` such as:

```
GET http://n1.org/call?goal=human(Who)&format=prolog
```

Because of the default values of the corresponding slicing parameters, the response in this case contains a complete slice with all solutions to `human(Who)` currently returned in one roundtrip. Since the URI specified `format=prolog`, this was the result:

```
success([human(plato),human(aristotle)], false)
```

At this point the implementation of `rpc/2-3` on `n2.org` must convert the returned slice into the intended nondeterministic behaviour. The usual mechanism is to keep the slice locally and enumerate it using `member/2`. This yields solutions to `human(Who)` on backtracking without further network communication, until the slice is exhausted.

Since the browser client still wants only one solution at a time, `n2.org` returns only the first derived solution, `Who=plato`, to the browser:

```
{ "type": "success",
  "data": [{"Who":"plato"}],
  "more": true }
```

When the user again types a semicolon, the browser requests another solution to `mortal(Who)`. In this particular run, no additional request to `n1.org` is needed: `n2.org` can simply backtrack within the local `member/2` enumeration of the slice already obtained from `n1.org`. The browser request therefore looks like:

```
GET http://n2.org/call?goal=mortal(Who)&offset=2&limit=1
```

and the response contains the next binding, `Who=aristotle`, now with `more:false`:

```
{ "type": "success",
  "data": [{"Who":"aristotle"}],
  "more": false }
```

This completes the description of what happens "under the hood" in the scenario of Figure 4.2: the browser drives one-solution-at-a-time enumeration through stateless HTTP requests to `n2.org`, while `n2.org` uses `rpc/2-3` to consult `n1.org` and reconstructs nondeterminism locally from slices of answers.

### Large result sets and the `limit` option

If `n1.org` had millions of clauses of `human/1`, returning a huge slice in one response would be undesirable. In that case `n2.org` can pass a smaller `limit` to `rpc/2-3` to retrieve answers in manageable batches:

```
human(Who) :-
    rpc('http://n1.org', human(Who), [
        limit(1000)
    ]).
```

This avoids swamping the caller with data. A goal with very many solutions will be delivered incrementally over multiple roundtrips, but since `rpc/2-3` is nondeterministic, the call remains logically complete provided the caller keeps asking for more.

As this scenario involves only two nodes, the Prolog Web here is tiny. Later we argue that the same principles can, in principle, scale to arbitrarily many nodes. Key is the combination of statelessness (for robustness and simplicity) with an execution model that avoids shipping excessive data and avoids recomputing answers unnecessarily.

### 4.2.5 Backtracking as pagination

The `offset` and `limit` parameters in the stateless HTTP API may look like a polite concession to web conventions – a way to make Prolog's multiple solutions fit a request–response protocol. But the connection is deeper than that. Prolog backtracking and HTTP pagination instantiate the same abstract idea: incremental enumeration under external control.

A paginated endpoint does not "return the result set"; it exposes a procedure for generating results and a protocol for consuming them in slices. The server produces a page, then stops. A later request asks for the next page, which amounts to resuming enumeration at a recorded position. Prolog behaves in the same general way. A goal may have many solutions; the toplevel presents the next one (or the next batch) and then waits. Internally, evaluation proceeds by creating choice points, and enumeration can be resumed from a well-defined position. In an interactive toplevel, that position is represented by retained search state, and the user's semicolon is simply the "resume" signal.

#### Offset surface, cursor semantics

Web API pagination comes in two principal forms. In *offset pagination*, the client says "skip $N$, return the next $k$"; the server must, at least logically, regenerate the first $N$ results and discard them. In *cursor pagination*, the server hands the client an opaque token that encodes where to resume; the client returns the token to request the next page, and the server picks up where it left off. The token is meaningless to the client – it is a handle to internal state, not a numeric position in a pre-existing result set.

Prolog backtracking is structurally cursor-based. A choice point is opaque retained state that captures exactly where to resume the search: variable bindings, unexplored branches, the position in the clause index. The semicolon hands it back and says "continue." No re-enumeration from the top occurs; no solutions are skipped and discarded.

On the wire, however, the stateless HTTP API speaks `offset` and `limit` because that is idiomatic REST:

```
GET /call?goal=hotel(Name,paris,Rating)&offset=20&limit=10
```

This returns solutions 21–30. But the implementation need not treat the offset as an instruction to re-enumerate from the beginning and discard. A node can cache a suspended toplevel process keyed by goal and offset, so that a request for page $k+1$ resumes the process from its suspended choice point rather than replaying the search. The offset then functions not as a distance from the start, but as a lookup key into retained state – in effect, a cursor in offset clothing.

### Consistency and the update view

The distinction between true offset re-execution and cached-process resumption is not merely one of cost; it has semantic consequences. True offset pagination – re-executing the goal fresh for each page – gives the client a sliding window over whatever the database contains at the moment of the request. If facts are asserted or retracted between pages, solutions may be duplicated, skipped, or reordered. Cursor-based resumption, by contrast, continues an in-flight enumeration whose view of the database is governed by the Prolog system's update semantics.

In the Prolog literature, this is the distinction between the *logical update view* and the *immediate update view*. Under the logical update view – which the ISO Prolog standard mandates – a goal sees the set of clauses that existed when it was called, regardless of later modifications. A cached toplevel process therefore provides snapshot-consistent pagination: each page continues a coherent enumeration even as the shared database evolves underneath. Under the immediate update view, the running goal would see mid-enumeration changes, weakening consistency but still preserving the search-tree position.

The choice of update view thus determines the isolation level of paginated enumeration, connecting a seemingly low-level implementation decision to the same consistency questions that web developers face when paginating over mutable data – and that database systems address with server-side cursors and versioned snapshots. A Prolog system that provides the logical update view delivers, essentially for free, the kind of stable pagination that web backends often must engineer with care.

### Three points on a spectrum

The relationship between Prolog's backtracking mechanism and HTTP pagination can be understood as a spectrum of three API styles, each exposing more of the underlying process model.

1. **Stateless offset, naive implementation.** Each request re-executes the goal against the current database, skips $N$ solutions, and returns the next batch. This is true offset pagination: cheap on server memory, idempotent by construction, but vulnerable to inconsistency if the database changes between pages.

2. **Stateless offset, cached processes.** The wire protocol still says `offset/limit`, but the server caches a suspended toplevel process keyed by goal and offset. Offset on the surface, cursor under the hood. Under the logical update view, this gives snapshot-consistent pagination while retaining a conventional REST interface.
3. **Semi-stateful API with process identifiers.** The server hands back an explicit process identifier, and the client uses it to request continuation. There is no offset at all. This is cursor pagination without disguise, and the closest structural match to interactive backtracking: the pid *is* the choice-point handle, and "next" *is* the semicolon.

The three styles differ in what the client's handle refers to. In (1), there is no handle; each request reconstructs everything from scratch. In (2), the offset is an implicit handle to a suspended search tree – choice points and bindings against the shared database. In (3), the pid is an explicit handle to a full process: search tree, variable bindings, *and the process's private database*, including any facts it has asserted locally or code loaded via `load_*` options. Successive "next" requests do not just advance enumeration; they resume a process that may have asserted local facts, updated counters, or built up intermediate structures as part of its computation. This is closer to a database session with a temporary table than to a mere cursor.

### When the stateless API grows stateful

The clean separation between stateless offset pagination and stateful pid-based resumption blurs once the stateless API accepts `load_*` options. A request such as

```
GET /call?goal=solve(X)&load_module=planner&offset=0&limit=5
```

causes the node to spin up a process that loads the specified module into its private database before executing the goal. That process now carries local state – the loaded clauses – not just a position in a search tree over the shared database.

If the node caches this process for subsequent page requests, the cached handle points to a process with a populated private database: essentially the same kind of entity that a pid refers to in the semi-stateful API. The only difference is how the client addresses it – an offset key derived from the goal and its options, versus an explicit pid.

If the node does *not* cache – naive re-execution – then each page request reloads the code, re-executes from scratch, and skips. This is not merely inefficient for enumeration; it re-runs the entire setup for every page. And if the loaded source has changed between requests, different pages may be enumerating over different programs entirely.

The conclusion is that even the API that looks most RESTful is, in the presence of `load_*` options, implicitly creating processes with private state. The question is only whether that state is retained across pages (cached) or reconstructed (naive), and whether the client gets an opaque handle to it (pid) or addresses it indirectly

(goal + options + offset as a composite key). The semi-stateful API simply makes this honest.

#### Caching and the web

The stateless design has a further consequence that is easy to overlook. Because Web Prolog pagination uses standard HTTP mechanics, it inherits the Web's caching infrastructure. A request for `?goal=mortal(X)&offset=0&limit=10` is a plain URI with a deterministic response. CDNs, reverse proxies, and browser caches can store and serve it without modification. Load balancers can route requests to any backend since there is no session state to preserve. And retry semantics are simple: the request is idempotent, so failures can be retried safely. None of this requires any special support from Web Prolog; it falls out automatically from the decision to expose backtracking as stateless pagination over HTTP.

These caching benefits apply most directly to the stateless API without `load_*` options – precisely the case where the offset is closest to a true numeric skip parameter and the response is most straightforwardly deterministic. As we move along the spectrum toward cached processes and `load_*` options, the caching story becomes more nuanced, but the surface-level HTTP compatibility is preserved.

### 4.2.6 Programming with `rpc/2-3`

Because `rpc/2-3` is a simple, synchronous, nondeterministic interface, it is easy to build small programming schemes on top of it. We sketch a few patterns here.

#### The parallel/1 predicate

A call to `parallel(Goals)` should

1. block until all work has been done, and no longer than necessary,
2. succeed, with variable bindings, if all goals succeed,
3. fail as quickly as possible if any goal fails, and
4. rethrow any errors thrown by a goal, also as quickly as possible.

In other words, running a list of goals in parallel should behave in the same way as when running them sequentially, but potentially complete sooner. For this to work properly, we must require something about the input too, namely that goals in the list are independent, that is, they must not communicate using shared variables or by any other means. From the caller's perspective, `parallel/1` is a Prolog-style behavior: a pure-looking meta-predicate whose implementation happens to orchestrate actors.

Crucially, `parallel/1` does not introduce a new observable interaction pattern. Provided the listed goals are independent and free of observable interference, the

call `parallel([G1, .., Gn])` is extensionally equivalent to sequential conjunction for deterministic goals.

That is, it succeeds with exactly the same variable bindings, fails under the same conditions, and propagates the same errors. The only intended difference is cost. In this sense, `parallel/1` is an optimisation of sequential conjunction rather than a semantic extension. Its internal use of actors and asynchronous messaging remains observationally invisible, and it therefore remains within the sequential partition of the Prolog Web.

### Making RPC calls in parallel

Two RPC calls in sequence take the running time of the first call plus the running time of the second:

```
?- rpc('http://n1.org', foo(X)),
   rpc('http://n2.org', bar(Y)).
```

If the caller node has ACTOR capabilities, we may combine RPC calls with `parallel/1`:

```
?- parallel([
       rpc('http://n1.org', foo(X)),
       rpc('http://n2.org', bar(Y))
   ]).
```

The time spent waiting is then approximately the maximum of the two response times. Note, however, that this is still a synchronous pattern: `parallel/1` blocks until both calls have returned. It therefore differs from actor-style interaction, where sends are asynchronous by default.

In its basic form, `parallel/1` commits to the first solution of each goal; it does not explore combinations by backtracking, so if `foo/1` or `bar/1` are nondeterministic, the above returns only one solution from each call. To aggregate lists of solutions, use `findall/3`:

```
?- parallel([
       rpc('http://n1.org', findall(X, foo(X), Xs)),
       rpc('http://n2.org', findall(Y, bar(Y), Ys))
   ]),
   aggregate_solutions(Xs, Ys, Solutions).
```

### Remote modules

Below we sketch an implementation of *remote modules*:[2]

[2] What we here call *remote modules* is similar in spirit to Ciao Prolog's *active modules*.

```
use_remote_module(URI, ImportList, Options) :-
    maplist(import(URI, Options), ImportList).

import(URI, Options, Functor1/Arity as Functor2) :- !,
    functor(Head1, Functor1, Arity),
    Head1 =.. [Functor1|Args],
    Head2 =.. [Functor2|Args],
    assertz((Head2 :- rpc(URI, Head1, Options))).
import(URI, Options, Functor/Arity) :-
    functor(Head, Functor, Arity),
    assertz((Head :- rpc(URI, Head, Options))).
```

Thanks to `load_*` predicates, an import directive can define local wrappers even for predicates not directly exported by the remote node, provided they can be expressed in terms of predicates that are exported.

### Ask around and seek agreement

```
ask_around(URIs, Query) :-
    member(URI, URIs),
    catch(rpc(URI, Query), _, fail).

seek_agreement([], _Query).
seek_agreement([URI|URIs], Query) :-
    catch(rpc(URI, Query), _, fail),
    seek_agreement(URIs, Query).
```

### Global goals

Our second example takes a step towards even more network transparency and is inspired by (Loke, 2006). The idea is to support *global goals* of the form `?- *Goal`, where a local peer database maps goal patterns to peers and options:

```
peer(foo(_), 'http://n1.org', [limit(10)]).
peer(bar(_), 'http://n2.org', [timeout(1),limit(100)]).
peer(bar(_), 'http://n3.org', [timeout(3),limit(100)]).
peer(baz(_), 'http://n3.org', [timeout(1),limit(50)]).
```

Given such a database,[3] `*/1` can be implemented like so:

```
:- op(100, fx, *).

* Goal :-
```

[3] We make no assumptions on how the peer database is updated.

```
    peer(Goal, Peer, Options),
    catch(rpc(Peer, Goal, Options), timelimit_exceeded, false).
```

A registry-based variation can be obtained by retrieving `peer/3` information remotely:

```
* Goal :-
    rpc('http://registry.pl', peer(Goal, Peer, Options)),
    catch(rpc(Peer, Goal, Options), timelimit_exceeded, false).
```

### 4.2.7 Promise and yield over HTTP

The `rpc/2-3` predicate is synchronous: the caller waits until the remote node returns an answer. For many applications this is fine, but sometimes a client wants to initiate a remote computation, continue with other work, and collect the result later. The `promise/3-4` and `yield/2-3` predicates support this split-phase pattern.

A call to `promise/3-4` sends a query to a remote node but returns immediately, handing back a reference that identifies the pending computation:

```
?- promise('http://n1.org', mortal(Who), Ref).
Ref = 19303011.
?-
```

The remote node begins evaluating `mortal(Who)`. Meanwhile, the caller is free to do other work. When it's ready for the answer, it calls `yield/2-3` with the reference:

```
?- yield($Ref, Answer).
Answer = success([mortal(plato), mortal(aristotle)], false).
?-
```

If the answer has already arrived, `yield/2-3` returns immediately. If not, it waits. The answer is delivered to the caller's mailbox, so `yield/2-3` must be called by the same process that issued `promise/3-4`.

The familiar options carry over: `template/1` controls which bindings are returned (and can reduce payload size via template packing, cf. Section **??**), while `offset/1` and `limit/1` control slicing. By default, `promise/3-4` requests all solutions; with `limit(K)`, the answer contains at most $K$ solutions, allowing large result sets to be fetched in batches. It is often useful to think of these batches as *pages* of an answer stream: `limit(K)` sets a page size, and `offset` selects which page to retrieve next. In Section 4.2.5 we will return to this idea and argue that pagination is best understood as a web-facing form of Prolog-style backtracking.

#### Bounded waiting and timeouts

A client that cannot afford to block indefinitely can specify a timeout:

```
?- promise('http://n1.org', expensive_query(X), Ref),
   yield(Ref, Answer, [timeout(2.0)]).
?-
```

If no answer arrives within two seconds, `yield/3` succeeds anyway – but `Answer` remains unbound. The caller can test for this and decide how to proceed: retry, report an error, or fall back to a default.

A polling pattern emerges naturally. The following code initiates an asynchronous call and then periodically checks for the answer while doing other work:

```
?- promise('http://n1.org', mortal(Who), Ref, [limit(100)]),
   repeat,
     yield(Ref, Answer, [
       timeout(0.2),
       on_timeout(writeln('Still waiting...'))
     ]),
     (  var(Answer)
     -> do_other_work, fail
     ;  !
     ).
Still waiting...
Still waiting...
Answer = success([mortal(plato), mortal(aristotle)], false).
?-
```

Each iteration waits briefly for an answer. On timeout, `on_timeout/1` is called and `yield/3` succeeds without binding `Answer`, allowing the loop to continue. Once the answer arrives, the cut terminates the loop.

This pattern – non-blocking initiation followed by pull-based answer consumption – preserves the sequential, declarative character of the Prolog Web while giving clients explicit control over timing.

This is one instance of a more general rule: on the Prolog Web, timing belongs in the protocol, at the points where a process would otherwise block.

## 4.3 The programmable Prolog Web

The Prolog Web is programmable in a stronger sense than the familiar 'Web of APIs'. In its original form as a hypertext Web of documents, the Web was not programmable at all. This changed early with server-side scripts that generated documents dynamically, and with client-side JavaScript. Later, remote procedure calling and open web APIs made it natural to treat the Web as a platform for composition, where independently developed services could be combined into "mashups", often using a client language as glue.

Web Prolog supports this conventional style of integration, but also generalises it. A node may offer clients not merely a fixed set of endpoints, but a Web Prolog runtime environment that can be programmed directly by queries of varying complexity and, when permitted, by the injection of source code into long-lived processes on the node. In addition, a node owner may install node-resident predicates that become part of what clients can call. In this sense, the Prolog Web is programmable both from the outside, by clients issuing queries and spawning processes, and from the inside, by node owners shaping the execution environment that clients interact with.

This dual perspective aligns naturally with the actor model. A node may restrict which capabilities are present in an actor at birth, for example via `load_*` options that provide an initial private program, and via placement choices (`node`) that determine in which environment the actor should live. Some behaviour is therefore "innate" to the actor, while other behaviour is acquired through interaction. Both are subject to the policy, resource limits, and trust assumptions of the hosting node.

Once code and data can cross node boundaries in either direction, a central systems question arises. Should a computation be moved to where the data resides, or should data (or code-as-data) be imported to where the computation runs? This question has no single correct answer. The optimal direction depends on data volume, update frequency, reuse patterns, latency requirements, and trust. The Prolog Web therefore does not enforce a fixed architecture, but instead exposes a small set of orthogonal mechanisms that let programmers choose, per interaction, how code and data should flow.

The owner's side of this exchange was illustrated in Section 4.1.4: installing predicates in the shared database and running node-resident actors that define what clients can call. The remainder of this section focuses on the client's side: how the `load_*` options let a client determine not only *where* to run, but *what* to run.

### 4.3.1 Injecting code with the load_* options

Looking back at the ping-pong scenario in Section 4.1.2, what if the definition of `pong/1` is available at `n1.org`, but not at `n2.org`? The `load_predicates` option ensures that the appropriate source code is shipped from the caller's database and injected into the private database of the actor that will run at `n2.org`:

```
ping_pong :-
    spawn(pong, Pong_Pid, [
        node('http://n2.org'),
        load_predicates([pong/1])
    ]),
    spawn(ping(3, Pong_Pid)).
```

The `load_predicates` option names predicates that already exist in the caller's context. When the clauses themselves should be supplied literally, the `load_list`

option serves that purpose. The following example uses it to populate a remote toplevel actor's private database with a clause that participates in the query:

```
?- toplevel_spawn(Pid, [
       node('http://n1.org'),
       load_list([(mortal(Who):-human(Who))])
   ]).
Pid = 83110912@'http://n1.org'.
?-
```

The query `?-mortal(Who)` now executes on `n1.org`, combining the injected rule with the node's own `human/1` facts:

```
?- toplevel_call($Pid, mortal(Who), [
       template(Who)
   ]).
?- receive({A -> true}).
A = success(83110912@'http://n1.org',[plato,aristotle],false).
?-
```

### 4.3.2 Code shipping and data locality

On the Internet, the cost of moving code and data across node boundaries is significant, yet unavoidable. The practical question is therefore not whether movement happens, but in which direction it should happen in order to reduce cost and improve responsiveness. In some cases it is more efficient to bring a small computation to a large or frequently updated dataset; in others, it is better to import data once and reuse it locally.

The most direct way to bring code to the data in Web Prolog is to inject a small, task-specific predicate into a process placed near the data, execute it there, and return only the derived result. This corresponds closely to the classical systems idea of *function shipping*: rather than transferring a large dataset across the network, one ships a small computation to where the data resides and retrieves only a compact summary.

Conversely, bringing data to the code can be expressed by running a goal locally while importing a remote program into the local process. This is where a unifying feature of the design becomes visible: since the default value of the `node` option for `spawn/1-3` and `toplevel_spawn/1-2` is the special-purpose URI `localhost`, the same `load_*` mechanisms used for remote execution also work for local code composition. The `load_*` options are not a "remote" feature; they are a general code-composition mechanism that happens to work transparently across the network. In this way, Web Prolog treats code and data symmetrically: facts and rules can be moved, cached, and reused much like datasets, reflecting Prolog's natural blurring of the distinction between code and data.

These examples suggest a useful symmetry. Web Prolog code can flow from client to node or from node to client, and the direction can be chosen declaratively by configuring the actor or toplevel used for the interaction. In principle, this choice can even be made programmatically at runtime, allowing applications to adapt their strategy based on observed latency, data size, or load.

### 4.3.3 A small design space

Taken together, the mechanisms described in this chapter form a small but expressive design space structured around a handful of recurring choices:

- **Interface**: `spawn`, `toplevel_spawn`, `rpc`, or `promise/yield`?
- **Placement**: locally or remotely (`node`)?
- **Code origin**: from the caller's database, shipped literally, or already resident on the target node (`load_*`)?
- **Interaction style**: pure query, interactive enumeration, or actor-style orchestration?
- **Synchrony**: should the caller block, or initiate work and collect the result later?
- **State**: stateless request–response, or session-oriented exchange?

The point is not that one combination is always best, but that the Prolog Web becomes programmable in a richer sense by letting developers decide where computation happens and how programs and data flow, rather than hard-coding those decisions into the architecture.

Code shipping is also the point where portability stops being an academic concern and becomes operational. If nodes can execute code originating elsewhere, then even small semantic differences or library mismatches become visible immediately. For this reason, the Prolog Web treats mobile code as profile-scoped: a piece of code should state, implicitly or explicitly, which profile it assumes, so that execution is either well-defined or rejected early.

In the limiting case, when a program is self-contained – all predicates it depends on are either included or guaranteed by the target profile – relocation becomes immediate. A client can download a program from one node and begin executing it locally, or ship it to another node, without any adaptation step. We refer to this property as *instant portability*. It is not a special mechanism but a direct consequence of the design: profile-scoped code, symmetric code flow, and the uniform treatment of local and remote execution combine to make relocation a deployment decision rather than a porting effort.

## 4.4 Timing and failure boundaries on the Prolog Web

Distributed execution differs from local execution in one decisive respect: waiting becomes observable. A goal that would either succeed or fail immediately in a local Prolog system may, when executed across a network, block for an indeterminate period. Timeouts therefore cannot be treated as mere transport parameters; they define the points at which partial failure becomes visible to the programmer.

In the Prolog Web architecture, two transports are used – HTTP and WebSocket – but three Web APIs are exposed: a stateless HTTP API, a semi-stateful HTTP API, and a stateful WebSocket API. Each API introduces a different blocking surface. Timeouts attach to those surfaces. Their meaning, and therefore their appropriate values, depend on the interaction model.

### 4.4.1 Three blocking surfaces

**Stateless HTTP API – request–response blocking.** Blocking occurs at the boundary of a single HTTP request. A client submits a goal and waits for a response. The timeout bounds that roundtrip. Because each call is isolated and resources are allocated per request, timeouts are typically short and strictly bounded. Pagination via `limit` and `offset` exists precisely to control cost and latency.

**Semi-stateful HTTP API – split-phase waiting.** Work submission and result retrieval are separated. A client initiates a task and later invokes `yield/3` with a `timeout/1` option. The timeout now bounds waiting for a previously initiated computation. Short polling intervals, retries, and fallback actions (`on_timeout/1`) become explicit application logic. Session-level idle or running limits govern resource reclamation independently of per-call waiting.

**Stateful WebSocket API – mailbox waiting.** Blocking occurs inside `receive/2`. The timeout bounds participation in an interaction protocol rather than a single computation. A mailbox wait may represent a reply, a coordination step, or a conversational turn. Its value is therefore determined by protocol design, not merely transport latency. Longer waits are natural in sustained dialogues, provided recovery behaviour is defined.

### 4.4.2 A comparative example

Consider the goal `route(gothenburg, paris, Route, Cost)`, which computes a travel route together with its cost.

**Stateless HTTP.** The client poses the goal and waits for answers. The timeout bounds the entire roundtrip. If exceeded, the call fails from the client's perspec-

tive. The interaction remains goal-centric: one request, one response. Expensive computations must be constrained (e.g. by limiting search) or retrieved in slices.

**Semi-stateful HTTP.** The client submits the goal but separates submission from waiting. The node may continue computing after the initial response. The client later calls `yield/3` with a short timeout, explicitly managing when and how long to wait. Waiting becomes programmable rather than implicit.

**WebSocket.** The client sends a message and waits in `receive/2`. The timeout now bounds participation in a dialogue rather than completion of a single call. On timeout, the client may retry, redirect, cancel, or renegotiate. The logical goal is unchanged, but the interaction context has shifted from evaluation to conversation.

The difference is therefore not in what is computed, but in where waiting is exposed and how failure is interpreted.

### 4.4.3 Late replies

Timeouts do not cancel the world. A computation may complete after a client has stopped waiting. The question of what happens to the result – whether it is discarded, logged, or delivered if the session remains alive – is a question about the *semantics of timing*: it determines what the client can observe and what guarantees the API provides. The Prolog Web makes these semantics explicit at the language level through options such as `timeout` and `on_timeout`, rather than hiding them inside transport configuration.

The complementary question – what happens to the *resources* that produced the late reply, and who is responsible for reclaiming them – belongs to lifetime discipline and is addressed in Section 4.5.3.

### 4.4.4 Design principle

A simple rule emerges:

> Place timeouts at the point where a process would otherwise block, and choose their values according to the interaction model – not merely according to transport defaults.

Under this discipline, timing becomes part of the architectural contract. The two transports – HTTP and WebSocket – remain implementation details, but the three Web APIs expose distinct failure boundaries. Making those boundaries explicit is essential for predictable behaviour on an open network.

## 4.5 Security on the Prolog Web

> Don't let your portal to the Prolog Web become a portal to your entire computer.
>
> *T-800 Model 101*

The Prolog Web is designed to be open: any client that knows a node's URI can submit goals for evaluation. Openness of this kind raises an obvious question: what makes it safe to let strangers run code on your node?

The answer has two components. Three mechanisms – *inexpressibility*, *invisibility*, and *containment* – define the *structural* security posture: the shape of what client code can express, what it can reach, and how long its effects persist. These mechanisms arise directly from the language design, and are developed in detail below.

The relevance of these mechanisms varies with the node profile. For profiles such as RELATION and ISOBASE, where interaction is stateless or request–response over HTTP, the dominant concerns are those of conventional web services: input validation, resource control, authentication, and transport protection. The full capability discipline underlying invisibility and containment becomes essential only for profiles that support long-lived sessions and bidirectional communication, such as ISOTOPE and ACTOR, where process identity and message passing cross node boundaries. In the remainder of this section, we focus on this richer setting, where the full interplay of the three mechanisms becomes visible.

A complete security story also requires *authentication* (to ground the distinction between client and owner), *resource governance* (to prevent abuse within the sandbox), and *transport security* (to protect communication on the wire). These are deployment concerns that Web Prolog inherits from standard web infrastructure; they are discussed briefly in Section 4.5.4.

The layered structure of these mechanisms is illustrated in Figure 4.3. The three lower boxes represent the structural security mechanisms provided by Web Prolog itself; the upper layers correspond to operational and deployment concerns.

### 4.5.1 Inexpressibility: the sandbox

Web Prolog adopts the same basic strategy as browser-based JavaScript: client-injected code runs in a constrained language subset. The language deliberately excludes file I/O, direct OS access, raw socket programming, and persistent storage. These capabilities, if needed, are provided by the host platform and mediated by the node owner through predicates and services that the owner chooses to install – they are not expressible by client-injected code.

This is the price of allowing strangers to run code on your machine: the code must be incapable of causing certain classes of trouble. The restriction is enforced at the language level, not by runtime checks on individual calls, which means that a

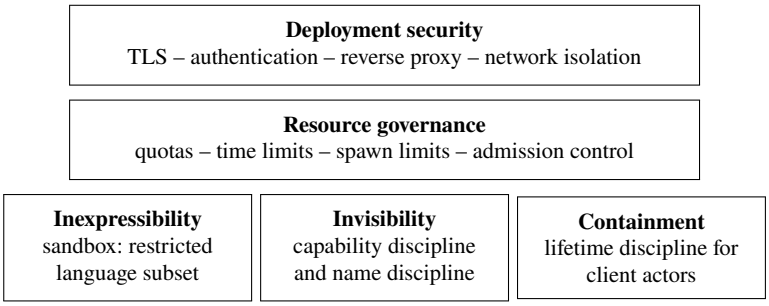



**Fig. 4.3** Layered security model of a Web Prolog node. The language sandbox restricts what client programs can express; capability discipline controls which actors and services they may reach; and lifetime containment ensures that client-created computation cannot persist beyond the session. Resource governance limits the amount of computation a client may perform, while deployment mechanisms secure communication and identity at the network level.

conforming implementation can reject dangerous code statically rather than catching violations at execution time.

Even when an operation exists in the underlying system, it is exposed to clients only through owner-controlled predicates and services, guarded by authentication and authorisation. The capability policy can therefore be read as part of the node's security boundary: it defines which actions are available to which role, and thereby what the node counts as its public surface.

### 4.5.2 Invisibility: name discipline

Joe Armstrong articulated the principle clearly: if we do not know the name of a process we cannot interact with it in any way, and the system is secure; once names become widely known, the system becomes less secure. Web Prolog inherits this principle directly.

A pid in Web Prolog is an opaque capability handle. The system does not provide any client-facing process-enumeration interface, and pids are chosen from a sufficiently large, unpredictable space that guessing a live pid is computationally infeasible. A client can therefore interact only with actors whose pids have been explicitly revealed to it, subject to the node's authorisation policy. A pid is a necessary capability – we cannot address what we cannot name – but not always sufficient; nodes may still require authorisation for operations performed via a known pid.

Discoverability creates a natural tension with name discipline. If processes are secure because you cannot talk to what you cannot name, then public services must still provide a way for clients to learn which names are meant to be used. The safe resolution is to treat discovery as an owner-curated *publication mechanism*: discovery should resolve *published* names, not reveal the existence of everything that happens

to be running. One concrete realisation of this idea – a declarative service directory queryable via `rpc/2-3` – was given in Section 4.1.4.

#### Reply channels as capabilities

A subtle but important consequence of treating pids as capabilities concerns the common request–reply idiom. In many actor systems, a client includes its own pid in a request so that the provider can send the reply. In a closed system this is usually harmless, but on the open Prolog Web it grants more authority than intended: a pid is not merely a return address, but a general communication capability.

Revealing a client pid therefore allows the provider not only to reply, but to send arbitrary messages to that actor. This creates a form of unintended capability delegation that may be benign in cooperative settings, but becomes problematic across trust boundaries.

A simple discipline avoids this form of capability leakage. Instead of exposing its own pid, a client should generate a fresh, request-local reply endpoint and include that in the request. Such an endpoint may be realised as a temporary proxy actor or equivalent restricted mechanism, and may be discarded after the reply has been received. In this way, the provider receives authority only to deliver the designated reply, not to invoke the client generally. In capability terms, the reply channel should carry only reply authority, not general send authority.

This pattern refines the capability discipline: communication authority should be as narrow and short-lived as the protocol step that requires it. This refinement is particularly important in distributed interactions, where reply channels cross node boundaries and therefore cross trust domains.

### 4.5.3 Containment: lifetime discipline

On an open network, a client may disconnect mid-session or vanish without warning. If server-side actors created on that client's behalf continue to run, they retain memory and send messages into mailboxes that may never be read. Over time, such orphans accumulate into unbounded state and potential security exposure.

The preventive measure is to make process lifetimes *containment-shaped*: every actor spawned by a client is lifetime-dependent on its parent, so that parent termination deterministically tears down the entire subtree. In Web Prolog, `link(true)` is the default; creating an independent actor requires an explicit `link(false)`, and that option is reserved for node owners. A client cannot detach work from its session.

Remote placement sharpens the point. A client may place a child on a remote node via `node(URI)`, but only under lifetime dependence. Allowing a remote detached child would turn `spawn/3` into a deployment primitive and make orphan prevention intractable under disconnects and partitions. For that reason, client-side remote spawn rejects `link(false)`.

Containment addresses the *resource* side of the late-reply problem identified in Section 4.4: when a client stops waiting, the computation it initiated may still be running. Lifetime discipline ensures that such processes are reclaimed when the session ends, regardless of whether their results were ever consumed. The *observable semantics* of what the client sees at the timeout boundary – whether a late reply is discarded, delivered, or never generated – is a timing concern and is treated in that section.

Should client-originated detached jobs be needed in the future, they should be introduced as an explicit abstraction – with quotas, lifetime parameters, and auditable handles – rather than as a general relaxation of the containment default.

### 4.5.4 Deployment concerns

The three structural mechanisms above define the security *architecture* of the Prolog Web, but they do not by themselves constitute a complete deployment story.

First, the node must establish *who* the caller is. Without authentication, the distinction between client and owner has no firm operational basis. Second, the node must enforce *resource governance*. A client may remain entirely within the sandbox and yet still exhaust CPU time, memory, process slots, or network bandwidth. Third, communication between client and node must be protected in transit. Without TLS or equivalent transport protection, a third party may observe or tamper with messages, undermining confidentiality and integrity.

Fourth, the runtime process itself must be constrained by the host environment. Even a correctly specified language sandbox offers limited assurance if the underlying process retains unrestricted filesystem or network access; host isolation – through operating-system-level confinement, WebAssembly runtimes, or WASI-based embeddings – removes entire classes of risk that the language boundary alone cannot prevent. Section C.2 develops the relationship between language sandboxing and host isolation in detail.

These concerns are not part of the Web Prolog language design as such. They belong to deployment, and are naturally addressed using standard web infrastructure and operational controls. A production node therefore requires both structural security and secure deployment.

What remains after these mechanisms have done their work is the node's public surface: the predicates it exports, the services it publishes, and the policies under which they may be accessed. A well-configured node should therefore present a small, explicit, and auditable interface: everything else should be either inexpressible, invisible, or contained.

## 4.6 Summary

This chapter developed the Prolog Web as the third component of the Prolog Trinity, extending Web Prolog from a language for single-node programming into a language for distributed systems.

The *concurrent Prolog Web* introduced network-transparent actor programming: spawning processes on remote nodes, sending messages across node boundaries, and managing lifetimes through links and monitors. Local and remote message passing differ only in latency, not in kind. Node-resident actors demonstrated how an owner can install durable, shared services, and a naming and discovery mechanism showed how such services can be advertised without compromising name discipline.

The *sequential Prolog Web* provided a disciplined request–response abstraction. The key construct, `rpc/2-3`, realises nondeterministic remote procedure calls with familiar Prolog semantics, implemented first over a remote toplevel actor and then, more practically, over stateless HTTP. The split-phase `promise/3-4` and `yield/2-3` predicates extend the pattern to deferred interaction. Although the sequential layer is built on top of the concurrent substrate, it presents a model in which mailboxes and scheduling remain invisible.

The *programmable Prolog Web* showed how code and data flow between client and node. The `load_*` options let clients inject task-specific predicates into remote execution contexts, while the same mechanisms work locally via the `localhost` URI, revealing code composition as a general facility rather than a remote-only feature.

A section on *timing and failure boundaries* made explicit where waiting becomes observable across the three Web APIs – stateless HTTP, semi-stateful HTTP, and WebSocket – and established that timeouts belong at blocking surfaces, chosen according to the interaction model rather than transport defaults.

Finally, a section on *security* asked what makes it safe to let strangers run code on a node, and answered in terms of three structural mechanisms – inexpressibility, invisibility, and containment – together with the deployment concerns (authentication, resource governance, and transport security) that complete the picture. What these mechanisms leave exposed is the node's *public surface*: the predicates it exports, the services it publishes, and the policies under which they may be accessed.

The chapters that follow build on this foundation. Part II develops generic behaviours – servers, supervisors, toplevels, and statecharts – that structure actor implementations, and later parts return to the architectural and paradigmatic questions that these technical layers raise.

# Part II
# Behaviors and other generics

# Chapter 5
# Implementing Web Prolog generics

Generic constructs provide reusable actor patterns that can be specialized, configured or extended post creation-time, and that encapsulate common concerns such as message protocols, supervision and state management.

It is useful to distinguish two broad categories of generics supported by Web Prolog. The first category comprises *Erlang-style behaviors*, which are message-driven components exposing a stable protocol, typically realised as a fixed vocabulary of messages and replies. They do not participate directly in the caller's proof search or bind the caller's logic variables. Their purpose is to provide robust, reusable services with well-defined communication patterns. The `server_*` predicates implement a message-driven request-response service analogous to Erlang's `gen_server` behavior, the `supervisor_*` predicates provide a behavior similar to Erlang's supervisor behavior, and statechart actors implement an actor-based state machine model in the spirit of Erlang's `gen_statem`. The `toplevel_*` predicates also fall into this category, even though the abstraction they support – a Prolog toplevel actor – would not ordinarily be found in Erlang.

The second category comprises what may be termed *Prolog-style generics*. Some of them are meta-predicates that internally orchestrate actors yet present a conventional predicate-level interface. From the programmer's perspective, they are ordinary predicates whose execution is implemented via actor orchestration behind the scenes. Examples include `parallel/1` and `first_solution/2-3`, which spawn worker actors, coordinate them with selective receive, and enforce particular success, failure, and error propagation semantics. They are semi-deterministic and do not themselves enumerate solutions upon backtracking. By contrast, `rpc/2-3`, introduced in Chapter 4, is a Prolog-style behavior that can succeed nondeterministically with multiple solutions, since it acts as a distributed analogue of a predicate call and therefore participates directly in proof search.

This separation between Erlang-style behaviors and Prolog-style generics reflects a dual identity of the language. On one hand, Web Prolog inherits the actor model, including explicit message passing and robust supervision. On the other hand, it maintains Prolog's commitment to declarative interfaces and transparent variable binding. Behaviors provide a structured means of reconciling these two strands:

they offer reusable actor frameworks where the machinery of communication, supervision, and concurrency is factored out from the application logic, and where programmers can work either at the level of actor protocols or at the level of ordinary predicates, depending on the needs of their application.

## 5.1 The server behavior: a preliminary sketch

The *server behavior* implements a classic client-server pattern in Web Prolog: a central server process maintains a private state and serves an arbitrary number of clients that communicate through message passing. The goal is to obtain a generic, reusable, and stateful server that can be specialised by providing an application-specific callback predicate, while also supporting hot code swapping of that callback without interrupting service or losing state. This is a prototypical Erlang-style behavior: it exposes a stable message protocol without binding the caller's variables.

Here is a small convenience wrapper for spawning a server:

```
server_spawn(Pred, State, Pid) :-
    server_spawn(Pred, State, Pid, []).

server_spawn(Pred, State, Pid, Options) :-
    spawn(server_loop(Pred, State), Pid, Options).
```

The predicate `server_spawn/3-4` takes the name of a callback predicate `Pred/4`, an initial state `State`, and spawns a new server process running `server_loop/2`. In this design, the state is always owned by the server process and never shared directly with clients. The behaviour of the server itself is captured by the following loop:

```
server_loop(Pred, State0) :-
    receive({
        '$call'(From, Ref, Request) ->
            call(Pred, Request, State0, Response, State),
            From ! Ref-Response,
            server_loop(Pred, State) ;
        '$upgrade'(Pred1) ->
            server_loop(Pred1, State0) ;
        '$stop'(From) ->
            From ! reply(true)
    }).
```

When the pattern `'$call'(From, Ref, Request)` matches an incoming message, `call(Pred, Request, State0, Response, State)` invokes the callback predicate `Pred/4`, where `State0` is the current server state and `State` is the updated state after processing the request. The server then replies to the client `From` with a message `Ref-Response` and recurs with the new state. All application logic lives

in `Pred/4`; the generic server merely routes requests, manages the state parameter, and returns responses.

The second message form `'$upgrade'(Pred1)` implements a very simple kind of hot code swapping. Upon receiving such a message, the server discards the old callback `Pred` and continues the loop with the new callback `Pred1`, while keeping the current state `State0` unchanged. In this way we can change the server's behaviour at runtime without stopping the process or reinitialising its state.

Here is the implementation of a suitable client API:

```
server_request(To, Request, Response) :-
    server_request(To, Request, Response, []).

server_request(To, Request, Response, Options) :-
    server_promise(To, Request, Ref),
    server_yield(Ref, Response, Options).
```

The predicate `server_request/3-4` behaves as a synchronous remote procedure call: it sends a request to the server and blocks until the matching reply is received or a timeout is triggered (depending on the `Options` passed to `receive/2` via `server_yield/3`). Under the surface, it is implemented in two stages, separating the sending of a request from the collection of the corresponding response.

First, `server_promise/3` sends the request and returns a fresh reference that will identify the response:

```
server_promise(To, Request, Ref) :-
    self(Self),
    make_ref(Ref),
    To ! '$call'(Self, Ref, Request).
```

Here `make_ref/1` creates a unique reference value, and the client process identifier `Self` is bundled with the request. Since the reference is sufficiently unique, multiple outstanding requests from the same client can safely be in flight at the same time; replies can arrive in any order and will still be matched correctly.

Second, `server_yield/2-3` waits for the reply tagged with the given reference:

```
server_yield(Ref, Response) :-
    server_yield(Ref, Response, []).

server_yield(Ref, Response, Options) :-
    receive({
        Ref-Response -> true
    }, Options).
```

The receive pattern `Ref-Response` ensures that only the response corresponding to the given reference is accepted; any other messages remain in the mailbox. In effect, this gives us a synchronous call interface on top of the underlying asynchronous message passing, with explicit correlation of calls and replies.

To specialise the generic server into a concrete service, we define an appropriate `Pred/4`. For example, in the refrigerator simulation, the callback `fridge/4` implements a tiny key-value store where the keys are food items and the value is a multiset of stored items:

```
fridge(store(Food), FoodList, ok, [Food|FoodList]).
fridge(take(Food), FoodList, ok(Food), Rest) :-
    select(Food, FoodList, Rest), !.
fridge(take(_Food), FoodList, not_found, FoodList).
```

Using this definition we can start a server, store some food, and later retrieve it using the synchronous `server_request/3`:

```
?- server_spawn(fridge, [], Pid).
Pid = 99491660.
?- server_request($Pid, store(milk), Response).
Response = ok.
?- server_request($Pid, take(milk), Response).
Response = ok(milk).
?-
```

For applications where asynchronous interaction is preferred, here is how to use `server_promise/3` and `server_yield/2`:

```
?- server_promise($Pid, store(meat), Ref).
Ref = 11025019.
% ... perhaps do something else here ...
?- server_yield($Ref, Response).
Response = ok.
?-
```

Upgrading a running server is just another message:

```
server_upgrade(To, Pred) :-
    To ! '$upgrade'(Pred).
```

A client that knows the server's `Pid` (or registered name) can call `server_upgrade/2` to replace the callback predicate. This operation is instantaneous from the client's perspective and does not wipe the server's state, which is exactly what we want from hot code swapping in a stateful service.

Now suppose we have developed a more efficient implementation `fridge2/4`, perhaps using a better data structure representing the state. As long as `fridge2/4` obeys the same interface `Request + OldState -> Response + NewState`, we can upgrade the server in place:

```
?- server_upgrade(Pid, fridge2).
true.
?-
```

The server process continues to run, all existing state (the current contents of the fridge) is preserved, and subsequent requests are handled by the new callback. This small example demonstrates the essence of the server behavior: a generic, reusable scaffold that provides state management, synchronous request-response communication, and lightweight hot code swapping, while leaving the application-specific logic to the callback predicate.

We terminate the server by calling `server_stop/2`, which sends it a `'$stop'` message:

```
server_stop(To, Reply) :-
    self(Self),
    To ! '$stop'(Self),
    receive({
        reply(Reply) -> true
    }).
```

### Monitoring and fail-fast server calls

The basic server protocol presented above supports synchronous request–response interaction by tagging each call with a fresh reference and waiting for a reply of the form `Ref-Response`. However, as written, a client may block indefinitely if the server terminates between receiving the request and sending the reply.

This is a natural place to use the `monitor/2` predicate introduced in Chapter 2. Before sending a synchronous request, the client installs a monitor on the server. The call then waits for *either* the expected reply `Ref-Response` *or* a `down` message indicating that the server has terminated. This yields a *fail-fast* call abstraction: if the server dies, the waiting client is immediately notified and can raise an exception or otherwise propagate failure.

Unlike the `monitor(true)` option, also introduced in Chapter 2, `monitor/2` is not atomic with process creation: it is a separate operation performed *after* the target process already exists. This means that a server could, in principle, terminate in the small gap between obtaining its pid and installing the monitor. Nevertheless, `monitor/2` remains essential in practice, precisely because it applies to actors that already exist – for example servers discovered through registration, servers started long before the current client, or services whose pid is obtained from another component. In such cases, spawn-time monitoring is simply not available, and `monitor/2` is the appropriate tool.

We implement fail-fast calls by extending `server_promise/3` to also return a monitor reference, and by extending `server_yield/2-3` to accept either a reply or the corresponding `down` message:

```
server_promise(To, Request, Ref, MonRef) :-
    self(Self),
    monitor(To, MonRef),
    make_ref(Ref),
```

```
    To ! '$call'(Self, Ref, Request).

server_yield(Ref, MonRef, Response, Options) :-
    receive({
        Ref-Response0 ->
            demonitor(MonRef),
            Response = Response0 ;
        down(MonRef, _Pid, Reason) ->
            throw(server_down(Reason))
    }, Options).
```

Consequently, `server_request/4` must then be implemented like this:

```
server_request(To, Request, Response, Options) :-
    server_promise(To, Request, Ref, MonRef),
    server_yield(Ref, MonRef, Response, Options).
```

The monitor reference `MonRef` is essential: it identifies which monitor instance produced the `down` message, and it allows the client to cancel the monitor once the reply has been received.

Here is what happens when something goes wrong:

```
?- server_spawn(fridge, [], Pid, [monitor(true)]).
Pid = 41168156.
?- server_request($Pid, sore(milk), Response).
ERROR: Unhandled exception: Unknown message: server_down(false)
?- flush.
Shell got down(41168156,41168156,false)
true.
?-
```

The server crashed because we accidentally requested `sore(milk)` rather than `store(milk)`. The current version of `fridge/4` could not handle this.

The server behaviour illustrates a principle that runs through the rest of this chapter and, more broadly, through the design of the Prolog Trinity as a whole. What we have done is cleanly separate the fixed scaffolding of a stateful concurrent service – its message protocol, state threading, monitoring, and support for code upgrade – from the application-specific logic, which is captured entirely in the callback predicate. The developer writes only the part that expresses the domain semantics; the generic server handles all the operational machinery.

It is worth pausing to compare this pattern to the simple refrigerator simulation in Chapter 2. There, we modelled the behaviour of a fridge directly as a Prolog relation over states. Here, we take exactly the same relational specification and plug it into a reusable server framework that gives it a process identity, encapsulated state, synchronous or asynchronous interaction, and hot code swapping. Nothing about the original Prolog style is lost. Developers can still write ordinary pure predicates to describe how a state evolves, but they now have the option of turning such predicates

into fully fledged concurrent services with minimal additional code. This is the distinctive promise of the approach: it elevates familiar Prolog programming into a concurrent and distributed setting without requiring a change in declarative style.

Of course, this preliminary sketch omits some of the practical details that a production-quality server behavior would need to address. Nevertheless, it already captures the core pattern that might be refined and extended later.

## 5.2 The supervisor behavior

Fault tolerance is essential in software applications because failures are inevitable in real systems, and a fault-tolerant design ensures that individual errors do not cascade into system-wide outages but are contained, recovered from, and rendered harmless to users and neighbouring components.

In Erlang, fault tolerance is achieved through supervision trees – hierarchies of actors that contain and isolate crashes. At the core of this model are *supervisors*, actors whose sole purpose is to start child actors, monitor them, and decide how to recover from crashes. Supervisors themselves can be supervised, yielding supervision trees of, in principle, arbitrary depth.

Underpinning this approach is the so-called *let it crash* principle: worker actors are not burdened with complex error handling but are allowed to crash when something goes wrong. The supervisor detects the crash through a `down` message containing a pid, identifies the crashed actor by this pid, and applies a predefined restart strategy. Erlang's restart policies are uniform and predictable. Depending on the severity and scope of the crash, a supervisor may do nothing, restart the crashed actor process (and perhaps its siblings), or terminate itself so that its own supervisor can take corrective action.

The implementation that follows is deliberately minimal: a one-for-one restart policy with a single global restart budget, enough to show how the actor primitives compose into a supervision structure but far short of a production-quality supervisor. The richer mechanisms that a full implementation would require – alternative restart strategies, per-child restart intensities, dynamic child management – are discussed at the end of this section. Our purpose here is not completeness but demonstration: to confirm that the same `spawn`, `monitor`, `exit`, and selective `receive` primitives that built the server behaviour in the previous section are sufficient to support fault-tolerant process management as well.

A supervisor consists of a *specific part*, which defines which child processes to manage and under what policies they are started and restarted, and a *generic part* – responsible for starting, linking, monitoring, and restarting children. In our implementation, the specific part is given by a list of child specifications and the generic part is captured by the supervisor loop. Each child specification carries an identifier, a start goal, and a restart policy: *permanent* children are always restarted, *transient* children are restarted only if they terminate abnormally, and *temporary* children are never restarted.

The architectural heart of the supervisor is its main loop:

```
supervisor_loop(ChildList, Count0) :-
    receive({
        down(_, Pid, true) ->
            remove_child(Pid, ChildList, ChildList1),
            supervisor_loop(ChildList1, Count0) ;
        down(_, Pid, Why) ->
            (   Count0 > 0
            ->  Count is Count0 - 1,
                restart_child(Pid, Why, ChildList, ChildList1),
                supervisor_loop(ChildList1, Count)
            ;   exit(gave_up)
            ) ;
        '$stop'(From) ->
            terminate(ChildList),
            From ! reply(true)
    }).
```

The loop waits in a `receive` call for `down` messages from monitored children. If the exit reason is `true` (normal termination), the child is simply removed from the child list. If the reason indicates a crash, and the restart budget has not been exhausted, the crashed child is restarted and its entry in the child list replaced with the new pid. If the budget is exhausted, the supervisor terminates itself – which, if it is itself supervised, allows a higher-level supervisor to take corrective action. Upon receiving a stop message, the supervisor shuts down all children in an orderly fashion.

Here is a brief example. Given three workers – one that blocks forever, one that terminates normally after three seconds, and one that crashes after five – we start a supervisor with a budget of three restarts:[1]

```
?- supervisor_spawn([
       child(steady,  (receive({})),   permanent),
       child(runner,  sleep(3),        temporary),
       child(crasher, (sleep(5), exit(an_error)), permanent)
   ], Pid, [budget(3), name(my_supervisor), monitor(true)]).
Pid = 10039133.
?-
```

The crasher terminates abnormally and is restarted. After three restart attempts the budget is exhausted and the supervisor gives up, terminating itself. The remaining child (the steady worker) is killed by the link cascade when the supervisor dies – no orphaned processes are left behind.

This sketch omits several mechanisms found in a production-quality supervisor. Erlang's OTP, for example, supports configurable restart strategies – `one_for_all` and `rest_for_one`, which dictate whether sibling processes must also terminate

[1] The option and child-spec forms used in this sketch (notably `budget/1` and `child/3`) are provisional. For the normative, implemented supervisor API, see Appendix A.

when a crash occurs – as well as per-child restart intensities with time windows (e.g. three crashes in ten seconds), and a robust API for dynamically adding or removing child specifications at runtime. None of these extensions would require new primitives. Each is implementable using the same `spawn`, `monitor`, `exit`, and selective `receive` that the sketch already relies on, and in each case the callback-based structure would remain: the supervisor's generic loop handles lifecycle management while the child specifications capture the application-specific topology. A complete implementation covering all three restart strategies, dynamic child management, restart intensity tracking, nested supervision, and cascade cleanup – together with a test suite of sixteen passing tests – is available on the book's companion page.[2] The point is not that this sketch is complete, but that the actor substrate introduced in Chapter 2 is sufficient to support supervision patterns of arbitrary sophistication. For a detailed account of OTP's full supervision capabilities, see the Erlang documentation.[3]

## 5.3 The toplevel behavior

The toplevel behavior, controlled by predicates such as `toplevel_spawn/1-2` and friends, must also be implemented. In our experience, once we have a complete implementation of all the Erlang-style primitives for concurrency and distribution, the implementation of the toplevel behavior and the built-in predicates for controlling it is fairly straightforward.

We begin with an implementation of `toplevel_spawn/1-2`:

```
toplevel_spawn(Pid) :-
    toplevel_spawn(Pid, []).

toplevel_spawn(Pid, Options) :-
    self(Self),
    option(target(Target), Options, Self),
    option(session(Continue), Options, false),
    spawn(session(Pid, Target, Continue), Pid, Options).
```

Before we proceed to implement `session/3` we shall demonstrate a couple of predicates that would make the task much easier. SWI-Prolog offers a buil-in predicate `findnsols/4` which provides a useful foundation for our implementation. It is somewhat similar to the standard `findall/3`, but expects an integer `Limit` in its first argument and will generate at most that many solutions. It is also nondeterministic, so on backtracking it will do it again. We borrow an example of its use from the SWI-Prolog manual:[4]

---

[2] `https://...`

[3] `https://www.erlang.org/doc/system/sup_princ.html`

[4] `https://www.swi-prolog.org/pldoc/doc_for?object=findnsols/4`

```
?- findnsols(5, I, between(1, 12, I), L).
L = [1, 2, 3, 4, 5] ;
L = [6, 7, 8, 9, 10] ;
L = [11, 12].
?-
```

Another SWI-Prolog library predicate `offset/2` will also prove useful.[5] Its purpose is to *skip* the first *n* solutions to a goal, i.e. the first *n* solutions are computed, but not collected. Here is an example of its use:

```
?- offset(10, between(1, 12, I)).
I = 11 ;
I = 12.
?-
```

Combining `findnsols/4` with `offset/2` allows us to implement a predicate `slice/5` capable of computing a *slice* of solutions to a goal:

```
slice(Goal, Template, Offset, Limit, Slice) :-
   findnsols(Limit, Template, offset(Offset, Goal), Slice).
```

However, we are looking for *answers*, rather than just slices of solutions. By wrapping a call to `slice/5` in a call to `call_cleanup/2` wrapped by a call to `catch/3` we arrive at a predicate `answer/5` capable of producing the four different forms of answer terms that we need:

```
answer(Goal, Template, Offset, Limit, Answer) :-
   catch(
      call_cleanup(slice(Goal, Template, Offset, Limit, Slice),
                   Det = true),
        Error, true),
   (   Slice == []
   ->  Answer = failure
   ;   nonvar(Error)
   ->  Answer = error(Error)
   ;   var(Det)
   ->  Answer = success(Slice, true)
   ;   Det = true
   ->  Answer = success(Slice, false)
   ).
```

Moving along to the implementation of `session/3` and the PTCP, consider again the statechart in Figure 5.1:
The most important part of the implementation of the PTCP are the three inner states **s1**, **s2** and **s3**. We shall ignore for a moment the outer state labelled PTCP and implement a session like so:

[5] https://www.swi-prolog.org/pldoc/doc_for?object=offset/2

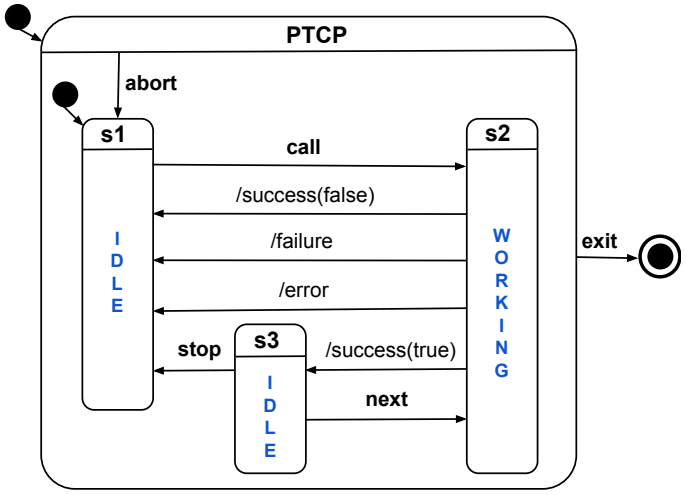



**Fig. 5.1** The complete Prolog Toplevel Communication Protocol.

```
session(Pid, Target0, Continue) :-
    state_1(Pid, Target0, Continue).
```

This takes us straight to **s1**, and here is what happens in the predicate `state_1`, which serves as the implementation of this state:

```
state_1(Pid, Target0, Continue) :-
    receive({
        '$call'(Goal, Options) ->
            option(template(Template), Options, Goal),
            option(offset(Offset), Options, 0),
            option(limit(Limit0), Options, 10 000 000 000),
            option(target(Target1), Options, Target0),
            Limit = count(Limit0),
            state_2(Goal, Template, Offset, Limit, Pid, Answer),
            Target = target(Target1),
            arg(1, Target, Out),
            Out ! Answer,
            (   arg(3, Answer, true)
            ->  state_3(Limit, Target)
            ;   true
            )
        }),
    (   Continue == false
    ->  true
    ;   state_1(Pid, Target0, Continue)
```

```
    ).
```

In `state_2` the real work is being done. This is where `answer/5`, defined earlier in this chapter, is used. However, answer terms must be extended with the pids of the actor processes that produced them.

```
state_2(Goal, Template, Offset, Limit, Pid, Answer) :-
    answer(Goal, Template, Offset, Limit, Answer0),
    add_pid(Answer0, Pid, Answer).
```

To handle this, `add_pid/3` is defined like so:

```
add_pid(success(Slice, More), Pid, success(Pid, Slice, More)).
add_pid(failure, Pid, failure(Pid)).
add_pid(error(Term), Pid, error(Pid, Term)).
```

One feature of `answer/5` that was not demonstrated before, is that the argument specifying the limit can be passed a term `count/1` with an integer in its argument. This works like a mutable local variable that can be assigned values using `nb_setarg/3` and read by means of `arg/3`.

```
?- Limit = count(2),
   answer(between(1,12,I), I, 0, Limit, Answer),
   nb_setarg(1, Limit, 5).
Limit = count(5),
Answer = success([1, 2], true) ;
Limit = count(5),
Answer = success([3, 4, 5, 6, 7], true) ;
Limit = count(5),
Answer = success([8, 9, 10, 11, 12], false).
?-
```

In the definition of the predicate `state_1/3` we saw that if a `success` answer term indicates (with `true` in its third argument) that there may be more solutions to the current goal, we enter `state_3`. For other answer terms a recursive call of `state_1/3` is made.

```
state_3(Limit, Target) :-
    receive({
        '$next'(Options2) ->
            (   option(limit(NewLimit), Options2)
            ->  nb_setarg(1, Limit, NewLimit)
            ;   true
            ),
            (   option(target(NewTarget), Options2)
            ->  nb_setarg(1, Target, NewTarget)
            ;   true
            ),
```

```
            fail ;
        '$stop' -> true
    }).
```

Here it is the reception of the `'$next'` message and the subsequent call to `fail/0` that triggers the backtracking to `answer/5` in state s2. If the `'$stop'` message is received instead, `state_3/2` terminates, and then `state_1/3` terminates too (unless the option `session(true)` was passed to `toplevel_spawn/1-2`).

As can be seen in the diagram depicting the PTCP, we have so far only implemented the three inner states.

We need to enable a client to abort the execution of a goal. We can do this by changing the implemention of `session/3` like so::

```
session(Pid, Target, Continue) :-
    catch(state_1(Pid, Target, Continue),
          '$abort_goal',
          session(Pid, Target, Continue)).
```

Here is how to tell the toplevel actor to abort the execution of any goal that it currently runs:

```
toplevel_abort(Pid) :-
    catch(thread_signal(Pid, throw('$abort_goal')),
          error(existence_error(_,_), _),
          true).
```

The action of aborting a particular execution of a goal passed to `toplevel_call/2-3` must not be confused with the action of exiting the toplevel process. The latter can be performed by calling `exit/1-2`.

Below, we implement `toplevel_call/2-3` simply as

```
toplevel_call(Pid, Goal) :-
    toplevel_call(Pid, Goal, []).

toplevel_call(Pid, Goal, Options) :-
    Pid ! '$call'(Goal, Options).
```

and `toplevel_next/1-2` like so

```
toplevel_next(Pid) :-
    toplevel_next(Pid, []).

toplevel_next(Pid, Options) :-
    Pid ! '$next'(Options).
```

and, finally, `toplevel_stop/1` like so:

```
toplevel_stop(Pid) :-
    Pid ! '$stop'.
```

## 5.4 The parallel/1 meta-predicate

In this section we will implement `parallel/1`, a meta-predicate that tries to run a list of goals in parallel. A call to `parallel(Goals)` should

1. block until all work has been done, but no longer,
2. succeed, with variable bindings, if all goals succeed,
3. fail as quickly as possible if any goal fails, and
4. rethrow any errors thrown by a goal, also as quickly as possible.

In other words, running a list of goals in parallel should behave in the same way as when running them sequentially, but finish faster. For this to work properly, we must require something about the input too, namely that goals in the list are independent, that is, they must not communicate using shared variables or by any other means. From the caller's perspective, `parallel/1` is a Prolog-style behavior: a pure-looking meta-predicate whose implementation happens to orchestrate actors.

To facilitate the explanation of our approach to implementing `parallel/1`, we first define a predicate `parallel/2` that accepts exactly two goals and spawns two actor processes to solve them in parallel:

```
parallel(Goal1, Goal2) :-
    self(Self),
    spawn((call(Goal1), Self ! Pid1-Goal1), Pid1),
    spawn((call(Goal2), Self ! Pid2-Goal2), Pid2),
    receive({Pid1-Goal1 -> true}),
    receive({Pid2-Goal2 -> true}).
```

The actor `Pid1` calls `Goal1` and, if it succeeds, sends the pair of `Pid1` and the (now instantiated) goal term as a message to the parent `Self`. The actor `Pid2` does the same thing for `Goal2`.

If the message sent by `Pid1` reaches the parent's mailbox first, then the first `receive` clause is triggered, and execution steps to the second `receive` statement and waits for the second actor's message to arrive. If the message sent by `Pid2` arrives first, then it is deferred. Once the message from `Pid1` comes along, the first `receive` statement is satisfied and the program steps to the second `receive` statement, which is triggered immediately by the deferred message. The result is that when `parallel/2` terminates, the messages will have been received irrespective of the order in which they were sent. The time spent waiting for the call to succeed is the longer of the response times from the two actors. As so often, asynchronous communication using send in combination with selective receive is key to the kind of programming going on here.

It is important to remember that the goal in the first argument of `spawn/1-3` is always copied before being called. This means that the instantiation of a goal passed to `parallel/2` will not happen until the corresponding call to `receive/1` has selected the message sent from that call.

This program satisfies our requirements 1) and 2), but not 3) and 4). The problem is that if one goal fails or throws an error, the corresponding `receive` will suspend,

waiting in vain for a message of the form `Pid-Goal` to show up. We will explain how to deal with this, but first show how our approach can be generalised into taking a list of goals instead of just two. For this, `maplist/3` comes in handy:

```
parallel(Goals) :-
    maplist(par_solve, Goals, Pids),
    maplist(par_yield, Pids, Goals).

par_solve(Goal, Pid) :-
    self(Self),
    spawn((call(Goal), Self ! Pid-Goal), Pid).

par_yield(Pid, Goal) :-
    receive({Pid-Goal -> true}).
```

This, by itself, does not help us satisfy the requirement 3) and 4), but here is a version of the program that does:

```
parallel(Goals) :-
    maplist(par_solve, Goals, Pids),
    maplist(par_yield(Pids), Pids, Goals).

par_solve(Goal, Pid) :-
    self(Self),
    spawn((call(Goal), Self ! Pid-Goal), Pid, [
        monitor(true)
    ]).

par_yield(Pids, Pid, Goal) :-
    receive({
        Pid-Goal ->
            receive({
                down(_, Pid, true) ->
                    true
            }) ;
        down(_, _, false) ->
            tidy_up_all(Pids),
            !, fail ;
        down(_, _, exception(E)) ->
            tidy_up_all(Pids),
            throw(E)
    }).
```

In this version, all three predicates from the previous program have been modified. In `par_solve/2` each actor running a call is now being monitored, and `par_yield/3` (which is `par_yield/2` with the addition of a third argument, the purpose of which we shall explain in a bit) is set up to inspect the `down` messages of the form `down(Ref,`

`Pid, Reason`) arriving from the worker actors and act appropriately. If `Reason` is `true` nothing needs to be done, if `Reason` is `false` the predicate `fail/0` is called as this will make `parallel/1` fail, and if `Reason` is of the form `exception(E)`, `E` is rethrown.

Note the use of an anonymous variable in the second argument of the patterns matching `false` and `exception(E)`. Using `Pid` here would work too, but might delay the failure of the call or the rethrowing of an error.

As soon as failure or error is detected, but before failing the entire call or rethrowing the error, the actor running `parallel/1` has a bit of tidying up to do, and this will be performed by a call to `tidy_up_all/1`. For this, it requires the complete list of worker actor pids, which has been passed along since its computation in the first call to `maplist/3` within the `parallel/1` definition.

The predicate `tidy_up_all/1` can be implemented as follows:

```
tidy_up_all(Pids) :-
    maplist(tidy_up, Pids).

tidy_up(Pid) :-
    demonitor(Pid),
    exit(Pid, kill),
    drain_mailbox(Pid).

drain_mailbox(Pid) :-
    receive({
        Pid-_ ->
            drain_mailbox(Pid) ;
        down(_, Pid, _) ->
            drain_mailbox(Pid)
    },[
        timeout(0)
    ]).
```

Here, one actor process at a time is killed by a call to `exit/2` (and recall that even if it is dead already, no error is raised). To avoid that the death of the process generates an additional `down` message, the monitoring of it must first be turned off. Finally, messages from the actor `Pid` that may have reached the mailbox during the execution of `parallel/1` are removed by a call to `drain_mailbox/1`.

It is time to test our program and make sure that we get a speedup and that our four requirements are satisfied. In the following call to `parallel/1`, we simulate goals with long running times by combining simple unifications with calls to `sleep/1`.

```
?- _Goals = [(X=a,sleep(1)),(Y=b,sleep(3)),(Z=c,sleep(2))],
   time(parallel(_Goals)).
 % 189 inferences, 0.000 CPU in 3.006 seconds
X = a,
Y = b,
```

```
Z = c.
?-
```

Here the call was done in just 3 seconds rather than the 6 seconds it would take if running them sequentially.

If one of the goals fail the entire call fails immediately, just as required by 3):

```
?- _Goals = [(X=a,sleep(1)),(Y=b,fail),(Z=c,sleep(2))],
   time(parallel(_Goals)).
 % 105 inferences, 0.000 CPU in 0.001 seconds
false.
?-
```

And finally, as required by 4), if one of the goals is bad an error is thrown that can be caught:

```
?- _Goals = [(X=a,sleep(1)),(Y=b,sleep(a)),(Z=c,sleep(2))],
   time(catch(parallel(_Goals),E,true)).
 % 126 inferences, 0.000 CPU in 0.001 seconds
E = error(type_error(float, a), context(system:sleep/1, _)).
?-
```

One thing to keep in mind is that if the goals in the list execute very quickly, the overhead of running `parallel/1` is likely to reduce any gains that might be had by running them in parallel. To make any noticeable difference, the predicate must be fed goals that run for a considerable time.

A predicate such as `parallel/1` is a valuable addition to the set of built-in predicates supported by Web Prolog, and it is encouraging to observe that it can be implemented with such succinctness. Naturally, there are multiple ways to implement this predicate. For example, consider Jan Wielemaker's implementation of `concurrent/3` (in his `library(thread)`[6]) which exhibits semantics sufficiently close to `parallel/1`. We are particularly encouraged by the fact that our implementation appears to be both simpler and more comprehensible than Wielemaker's. Code simplicity and clarity are important considerations, and code size matters.

## 5.5 The first_solution/2 meta-predicate

Another example of parallel processing supported by Wielemaker, also available in `library(thread)`, is `first_solution/2-3`. It tries alternative solvers in parallel, and as soon as one of the alternatives is successful, all the others are terminated.

Borrowing an example from Wielemaker, if it is unclear whether it is better to search a particular graph breadth-first or depth-first, we can use:

---

[6] https://www.swi-prolog.org/pldoc/doc/_SWI_/library/thread.pl?format_comments=true&show=src#concurrent/3

```
search_graph(Graph, Path) :-
    first_solution(Path, [ breadth_first(Graph, Path),
                           depth_first(Graph, Path)
                         ],
                   []).
```

Applying roughly the same pattern as in the implementation of `parallel/1`, we implement `first_solution/2` in Web Prolog below (leaving out the options that Wielemaker's version support):

```
first_solution(Solution, Goals) :-
    maplist(first_solve(Solution), Goals, Pids),
    wait_first(Pids, Solution).

first_solve(Solution, Goal, Pid) :-
    self(Self),
    spawn((call(Goal), Self ! Pid-Solution), Pid, [
        monitor(true)
    ]).

wait_first([], _) :- !, fail.
wait_first(Pids, Solution) :-
    receive({
        _ - Solution ->
            tidy_up_all(Pids) ;
        down(_, Pid, false) ->
            select(Pid, Pids, Rest),
            wait_first(Rest, Solution) ;
        down(_, _, exception(Error)) ->
            tidy_up_all(Pids),
            throw(Error)
    }).
```

Note the reuse of `tidy_up_all/1` from the implementation of `parallel/1`.

Using a simple simulation of two long-running goals, here is a trial run that shows that our predicate works as intended:

```
?- time(first_solution(X, [(sleep(2),X=a),(sleep(1),X=b)])).
 % 164 inferences, 0.000 CPU in 1.006 seconds
X = b.
?-
```

## 5.6 Summary

Taken together, the examples in this chapter illustrate how Web Prolog can factor recurring concurrency and coordination patterns into reusable behaviours. Erlang-style behaviours such as the server, supervisor, and toplevel wrap actors in explicit protocols and runtime scaffolding for state management, request–response handling, and fault tolerance, without binding the caller's variables. Prolog-style generics such as `parallel/1` and `first_solution/2` show how the same actor substrate can be used to implement higher-level meta-predicates that look and feel like ordinary Prolog, yet exploit parallelism and robust error propagation internally.

A single observation ties these examples together. In every case, the application-specific logic – the callback predicate that defines a server's domain semantics, the goals passed to `parallel/1`, the alternative solvers handed to `first_solution/2` – is written in ordinary Prolog: pure predicates, relational specifications, familiar recursive definitions. The concurrency, the fault handling, the protocol machinery, and the cleanup discipline are supplied by the generic framework. The developer writes the part that requires domain knowledge; the behaviour provides the rest. This separation is not merely a convenience. It means that the large existing body of Prolog programs and programming techniques does not need to be abandoned or rewritten in order to participate in a concurrent and distributed setting. It can be reused directly, wrapped in the appropriate behaviour, and deployed as a networked service.

But the chapter also reveals a less obvious point. The Erlang-style behaviours – server, supervisor, toplevel – have direct counterparts in Erlang's OTP, and Web Prolog's versions are broadly comparable in structure and length. The Prolog-style generics tell a different story. The implementation of `parallel/1` requires roughly thirty lines of code; `first_solution/2` requires fewer than twenty. Both are shorter and, we believe, more legible than their Erlang equivalents. The reason is not a single feature but a combination: `maplist` handles the orchestration loop, unification-based selective `receive` correlates replies without manual bookkeeping, goals-as-terms pass work units to spawned actors without serialisation or callback registration, and `fail` triggers backtracking cleanup naturally. These are ordinary Prolog mechanisms, none of them designed for concurrency, yet they compose fluently with the actor primitives to yield concurrency abstractions that would require considerably more scaffolding in a language without them. If the Erlang-style behaviours show that Web Prolog can match OTP's repertoire, the Prolog-style generics show where it can surpass it – and that difference is the clearest evidence this chapter offers for why the combination of logic programming and actor programming is more than the sum of its parts.

# Chapter 6
# The statechart behavior

The earlier chapters introduced several generic behaviors for actors in Web Prolog: the toplevel behavior, the server behavior, and the supervisor behavior. Each of these patterns offered a reusable programming model for structuring how actors communicate, manage state, or enforce reliability. In this chapter we introduce a more expressive and visually oriented behavior: the *statechart behavior*, which provides a powerful mechanism for defining how a single actor responds to events and evolves behavior over time.

A statechart behavior encapsulates the dynamics of an actor in terms of a state machine with hierarchy, parallel regions, and event-triggered transitions. This model is inspired by David Harel's statecharts, and the result is an abstraction capable of implementing interactive interfaces, conversational flows, control systems, and long-running reactive processes in a structured and compositional manner.

The representation we use in this chapter is based on a compact XML notation. Although XML is not always a popular choice as a general-purpose representation format, it fits statecharts unusually well because its tree structure mirrors hierarchical nesting of states and regions directly. The notation is directly executable, allowing statechart documents to be loaded, spawned, and controlled as Web Prolog actors. For example, a statechart can be launched with `statechart_spawn/1-2`, participate in supervision trees, and send or receive messages like any other actor.

## 6.1 Statecharts

Invented by David Harel, Statecharts is a graphical formalism for specifying reactive systems (Harel, 1987). Building on the concept of a state machine, it adds hierarchy, parallel regions (orthogonality), and event visibility across the active states and regions. Like logic, it is a "tool for thinking" with a formal semantics (Harel and Naamad, 1996), but it is also explicitly visual – a way of "tapping the potential of high bandwidth spatial intelligence, as opposed to lexical intelligence used with textual information" (Samek, 2002).

Statecharts have been used in a wide range of areas, from embedded control and user interfaces to conversational systems and protocols (Samek, 2002; Horrocks, 1999; Skantze and Moubayed, 2012; Cicirelli et al., 2009). In the Web context, the W3C standardized SCXML as a way to represent statecharts in an XML form (Barnett et al., 2015). SCXML has seen industrial use (for example by Genesys[1]). Ideas from statecharts are also visible in front-end libraries such as XSTATE, which claims SCXML compatibility while expressing machines in JavaScript.[2] For the back-end, *Statifier* is an Elixir implementation of SCXML with a focus on W3C compliance.[3]

In this chapter we implement most of the SCXML 1.0 processing model, but present a lightweight textual profile that is directly executable as a Web Prolog actor behavior, with Prolog as both data model and executable content. The surface syntax resembles SCXML, but it is intentionally not a full SCXML language binding; our goal is to keep the standard execution model while harmonizing it with Web Prolog and the Prolog Web.

Importantly, the exposition in this chapter is self-contained: readers do not need any familiarity with SCXML. Although our syntax resembles SCXML superficially, the mechanisms are presented here as a standalone behavior for Web Prolog actors. A comparative discussion of SCXML 1.0, and of Erlang's `gen_statem` behavior, is postponed until the end.

## 6.2 Statechart actors

A *statechart actor* is a Web Prolog actor whose control logic is given by a statechart document. Conceptually, it behaves like an ordinary actor with a mailbox, except that incoming messages are interpreted as *events* that drive a structured state machine rather than an ad hoc `receive/1-2` loop.

Events are just messages: any incoming actor message (a Prolog term) can serve as an external event. When an event is processed, it is *visible* to all currently active parts of the machine (including parallel regions), and it may trigger one or more transitions whose actions run as Web Prolog code.

Execution follows the standard statechart *run-to-completion* intuition: processing one external event may trigger a finite cascade of internal work (such as taking condition-only transitions and handling internally raised events) before the actor consumes the next external message. This makes each reaction behave like an atomic control step at the statechart level, even though the actor itself remains an ordinary concurrent process.

In the examples below, a statechart document can therefore be read as a compact description of (i) which states are active, (ii) which incoming events matter in

[1] See `https://docs.genesys.com/Documentation/OS`

[2] `https://github.com/statelyai/xstate`

[3] `https://riddler.github.io/statifier/`

those states, and (iii) what effects and state changes occur when such an event is observed. Precise processing rules are standardized by SCXML; see the W3C Recommendation, especially Appendix D (Algorithm for SCXML Interpretation) and Section 3.13 (Selecting and Executing Transitions).

### 6.2.1 A playful intro to statecharts

In its simplest form, a statechart is just a deterministic state machine, in which state transitions are triggered by events. The simple statechart in Figure 6.1 serves as our first example. The labelling of one of the transitions with the symbol `play` suggests that the example is drawn from the field of game development.[4]

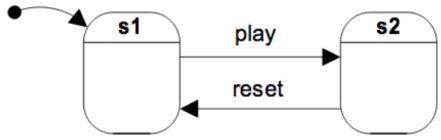



**Fig. 6.1** A minimal statechart example: event-triggered transitions in a flat machine.

Here, `s1` is the initial state. If the next selected event is `play`, a transition to `s2` is taken. For any other event, the machine will remain in `s1` and the event will be lost (i.e. it will not be deferred).

Any statechart can be translated into a document written in an XML-based syntax. Here, a document that captures the statechart in Figure 6.1 is shown:

```
<statechart datamodel="web-prolog" initial="s1">
    <state id="s1">
        <onentry>
            output('IDLE')
        </onentry>
        <go to="s2" on="play"/>
    </state>
    <state id="s2">
        <onentry>
            output('PLAYING')
        </onentry>
        <go to="s1" on="reset"/>
    </state>
</statechart>
```

Note that two `<onentry>` elements, the significance of which is not shown in Figure 6.1, have been added to the document. They contain *executable content* –

[4] We are borrowing some of our examples from (Brusk and Lager, 2008).

instructions which allow the state machine to *do* things such as modifying its data model or interacting with external entities. Executable content must be specified using Web Prolog source code. In the case of the above example, the use of the Web Prolog built-in predicate `output/1` ensures that an event message – either `'IDLE'` or `'PLAYING'` – is returned to the spawning process each time `s1` or `s2` is entered. Such code – which may also appear as the contents of <onexit> and <go> elements – must succeed deterministically, and if an error is thrown by such code, an internal error event will be sent, not to the mailbox of the underlying actor, but to an *internal* queue, where it may trigger transitions in the state machine.

### 6.2.2 A statechart process is an actor in disguise

A major feature of our proposal is that we encourage people to think of *a statechart process as an actor running on the Prolog Web*, either *in* a browser, or as a process running on a node and accessed *from* a browser. Indeed, to create a statechart session, we can simply use `statechart_spawn/2`:

```
?- statechart_spawn(Pid, [
       load_uri('http://n6.org/game.xml'),
       monitor(true)
   ]).
Pid = 78230912.
?- flush.
Shell got 'IDLE'
true.
?-
```

When `statechart_spawn/1-2` is called, a new actor is spawned that interprets the statechart. As specified by the <onentry> element in the initial state, the statechart actor immediately sends the event `'IDLE'` to its parent. Equipped with the pid, we can use `!/2` to send an event to the actor, causing a transition from `s1` to `s2`. In general, an event may be any Web Prolog term except a bare variable; here, the relevant `on` attribute tells us that the atom `play` will do:

```
?- $Pid ! play.
true.
?- receive({What -> true}).
What = 'PLAYING'
?-
```

We may now send the event `reset` to return to `s1`, and so on. At any point we may also terminate the actor by sending it a signal with `exit/2`. Since the `monitor` option was set to `true`, the actor delivers a `down` message to its parent just before it terminates, exactly as an ordinary actor would.

Other options to `statechart_spawn/2` work as expected. For example, we can run the statechart on a remote node by passing a `node(URI)` option, and we can provide the source directly using `load_text` rather than fetching it from a URI.

The examples in the rest of this chapter are organized around a small set of extensions that make statecharts practically useful as an actor behavior on the Prolog Web. Each extension adds a distinct capability – some at the state-machine level, others at the level of interaction with the surrounding actor system. Concretely, the extensions we rely on are:

- Hierarchy and history states
- Parallelism (a.k.a. orthogonality)
- Broadcast communication
- Process invocation
- Inter-process communication
- Data model and scripting language

In the sections that follow, the left column highlights statechart-level structuring mechanisms, while the right column highlights how those mechanisms connect to actors and Prolog in the Web Prolog setting.

## 6.3 Hierarchy and history

Statecharts may be *hierarchical*, i.e. a state may contain another statechart down to an arbitrary depth. A complex state may contain a *history state*, serving as a memory of which substate S the complex state was in, the last time it was left for another state. Transition to the history state implies a transition to S.

The statechart in Figure 6.2 exemplifies both hierarchy and the history state by enhancing our simple game with a pause-and-resume functionality.

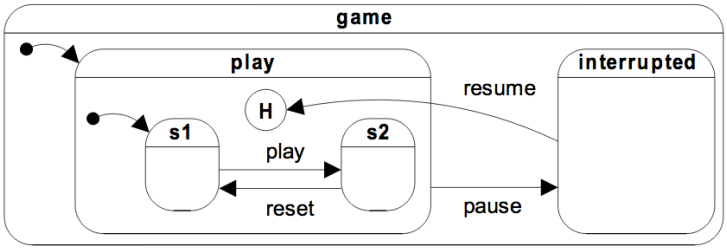



**Fig. 6.2** Pause-and-resume via hierarchy and history state in the game controller.

Suppose that the current state is `s2`, and that the next selected event is `pause`. The transitions leaving `s2` are tried from the inside and out, and since `reset` does not match the first event in the queue, but `pause` does, a transition to the `interrupted` state takes place. If a `resume` event then shows up in the queue, the system transitions to the history state `H`, which by the statechart semantics implies a transition back to `s2` again.

Documents are structured so that they mirror the topology of the statecharts they represent. A document that captures the statechart in Figure 6.2 is shown below:

```
<statechart datamodel="web-prolog" initial="Play">
    <state id="Play" initial="S1">
        <history id="H">
            <go to="S1"/>
        </history>
        <state id="S1">
            <go to="S2" on="play"/>
        </state>
        <state id="S2">
            <go to="S1" on="reset"/>
        </state>
        <go to="Interrupted" on="pause"/>
    </state>
    <state id="Interrupted">
        <go to="H" on="resume"/>
    </state>
</statechart>
```

Hierarchy supports two complementary design moves. In *refinement* (top-down), an abstract control state is elaborated into substates whose internal transitions make the behaviour explicit. In *clustering* (bottom-up), a family of closely related states is grouped under a superstate, so that shared transitions and common entry/exit behaviour can be expressed once. Both moves reduce state explosion and make large machines readable: the top level can remain a small "map" of the application, while local detail is pushed down into the regions where it belongs. Together, hierarchy and history give *interruption* and *resumption* a useful first-class representation.

## 6.4 Parallelism and broadcast communication

A hallmark of statecharts is *orthogonality*: a state may contain two or more regions that execute in parallel. Operationally, the machine is simultaneously in one active state in each region. The point is not CPU-level parallelism, but a way to factor one reactive controller into several facets that evolve together.

### 6.4.1 A slightly richer orthogonality example: an NPC controller

To see why orthogonality matters, consider a toy controller for a non-player character with two independent emotion dimensions and one behavior mode. Figure 6.3 sketches the idea: the character is simultaneously in one state in each emotion dimen-

sion and in one behavior state. When an event is processed, it may trigger coordinated changes across several regions.

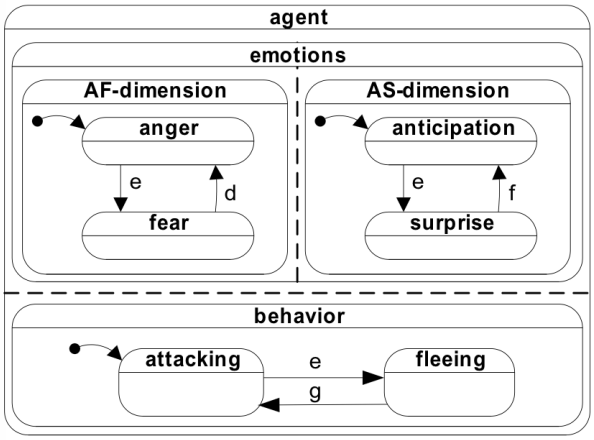



**Fig. 6.3** Orthogonal regions: two emotion dimensions plus one behaviour mode in an NPC controller.

In the figure, an `e` event (think: an explosion) moves the character into `fear`, `surprise`, and `fleeing` "at once", because all three regions observe the same event while they are active. Without parallel regions, we would be forced to encode combinations such as `anger_anticipation_attacking`, `fear_surprise_fleeing`, and so on, and the number of states grows quickly as more dimensions are added.

Here is how to render this machine in XML:

```
<statechart datamodel="web-prolog" initial="agent">
    <parallel id="agent">
        <parallel id="emotions">
            <state id="AF-dimension" initial="anger">
                <state id="anger">
                    <go to="fear" on="e"/>
                </state>
                <state id="fear">
                    <go to="anger" on="d"/>
                </state>
            </state>
            <state id="AS-dimension" initial="anticipation">
                <state id="anticipation">
                    <go to="surprise" on="e"/>
                </state>
                <state id="surprise">
                    <go to="anticipation" on="f"/>
```

```
                </state>
            </state>
        </parallel>
        <state id="behavior" initial="attacking">
            <state id="attacking">
                <go to="fleeing" on="e"/>
            </state>
            <state id="fleeing">
                <go to="attacking" on="g"/>
            </state>
        </state>
    </parallel>
</statechart>
```

### 6.4.2 A minimal broadcast example: two regions exchanging events

To make the mechanism completely explicit, we now switch to a deliberately minimal example: two regions that exchange events. The point is not to model two independent actors, but to show how a single statechart step can coordinate activity across regions.

```
<statechart datamodel="web-prolog" initial="Start">
    <parallel id="Start">
        <state id="Pinger">
            <onentry>raise(pong)</onentry>
            <go on="ping">
                log('Ping'), raise(pong)
            </go>
        </state>
        <state id="Ponger">
            <go on="pong">
                log('Pong'), raise(ping)
            </go>
        </state>
    </parallel>
</statechart>
```

Note the use of targetless <go> elements (i.e. elements without a `to` attribute). These transitions consume an event (and may run executable content, such as `raise/1`) without changing the active configuration, so no <onentry> or <onexit> code is triggered.

### 6.4.3 What "broadcast" means here

The key semantic idea is the *configuration*: the set of active states (and regions) that the machine is currently in. Whenever the interpreter selects an event for processing, that event is matched against transitions in *all* active states in the configuration. In this sense the event is "broadcast": it is visible everywhere in the current machine, even though only some region(s) react to it. Any internal events raised while handling the current input are processed before the next external message is consumed, so the effects within one reaction are deterministic given the current configuration, the data model, and the selected event.

### 6.4.4 Why not model each region as a separate actor?

A natural objection is that the Prolog Web already provides actor-level concurrency: why not represent each region as its own actor supervised by a parent, and coordinate by ordinary messaging? This is certainly possible, and it is often the right choice when true independence, failure isolation, distribution, or distinct lifetimes are required.

Orthogonality, however, is not primarily a claim about *parallel execution*; it is a way to specify a *single* reactive controller with an atomic, run-to-completion reaction. In a statechart, one selected event is visible to all active regions, transitions are chosen under a single set of priority and conflict-resolution rules, actions run, and the machine commits one coherent update of configuration and data model. An actor-only encoding can approximate the same effect, but typically only by introducing an explicit coordination protocol (disseminate, barrier, commit) and a coordinator that reimplements the statechart scheduler in ad hoc code, while also making cross-region invariants harder to maintain under message interleavings.

A simple design rule of thumb follows: use orthogonal regions when the facets are tightly coupled parts of one control process and deterministic, atomic reactions are desirable; use separate actors when the facets should be independently schedulable, restartable, or distributable.

## 6.5 The Web Prolog data model

For many applications, a statechart by itself does not suffice. For example, while statecharts may be able to handle reactive aspects of AI-related applications, proactive aspects such as planning, decision making, and many other forms of problem solving and "intelligent" behavior must also be catered for. Therefore, sophisticated conversational systems require knowledge representation and reasoning capabilities as well as interactional capabilities. In its current form, statecharts do not provide developers with sufficient means for knowledge representation and reasoning. To enable reactive control to be combined with logical inference, rule-based decision-

making, and symbolic computation, Web Prolog can be used for data modelling and scripting.

So far, every transition has reacted to an external event. Statecharts also support *eventless* transitions: transitions without an `on` attribute that may fire whenever their guard condition holds in the current configuration. Combined with a private and dynamic Prolog data model, this gives statecharts a form of forward-chaining inference: the machine repeatedly applies condition-action rules until no more apply, then waits for the next event. Here is a simple example, implementing Euclid's algorithm for calculating the greatest common divisor of a set of numbers:

```
<statechart datamodel="web-prolog" initial="Run">
    <datamodel>
        :- dynamic int/1.
        int(25). int(10). int(15). int(30).
    </datamodel>
    <state id="Run">
        <go if="int(X), int(Y), X > Y">
            Z is X-Y,
            retract(int(X)),
            assert(int(Z))
        </go>
        <go to="Stop" if="int(X)">
            writeln(X)
        </go>
    </state>
    <final id="Stop"/>
</statechart>
```

The example illustrates another important piece of statechart semantics: if the children of an active state contain more than one `<go>` element, they are tried in document order. In this case, if the order between the `<go>` elements was switched, the statechart would just write `25` (instead of the correct `5`) and then transition to the final state `Stop`, and terminate.

## 6.6 Process invocation and communication

Statecharts become much more useful once they can spawn other processes and *communicate* with them. In the Web Prolog setting, the invoked processes are simply actors: toplevels, other statechart actors, or services running on remote nodes. For example, the statechart below spawns a toplevel, submits a query, prints each answer as it arrives, and terminates when the last answer has been received.

```
<statechart datamodel="web-prolog" initial="spawn-ask-collect">
    <state id="spawn-ask-collect" initial="ask">
```

```
        <spawn type="toplevel"
               node="n9.org"
               session="false">
            p(a). p(b). p(c).
        </spawn>
        <state id="ask">
            <go to="collect" on="spawned(Pid)">
                toplevel_call(Pid, p(X), [
                    limit(1)
                ])
            </go>
        </state>
        <state id="collect">
            <go to="collect" on="success(Pid,Data,true)">
                writeln(Data),
                toplevel_next(Pid)
            </go>
            <go to="f" on="success(_,Data,false)">
                writeln(Data)
            </go>
        </state>
    </state>
    <final id="f"/>
</statechart>
```

**How to read the example.**

- The <spawn> element requests that a child actor be created when the surrounding state becomes active. Here the child is a toplevel actor that is started on the node given by `node="n9.org"`.
- The event `spawned(Pid)` reports the pid of the spawned actor. The transition in state `ask` reacts to this event by calling `toplevel_call/3`, which initiates query evaluation.
- Each answer arrives as an event of the form `success(Pid,Data,More)`. While `More` is `true`, the statechart requests the next slice of answers by calling `toplevel_next/1`. When `More` is `false`, the last answer has been delivered and the machine transitions to the final state.

Note that variables bound by matching the `on` pattern and by evaluating the `if` guard are in scope in the executable content of the same <go> element.

Also note that the intent of <spawn> is not merely to create a child process, but to tie its lifetime to control state: when the state that contains the invocation is exited, the invoked actor is terminated as part of leaving that state. This makes it easy to express resource ownership (“this control state owns this helper process”) without relying on ad hoc cleanup code.

## 6.7 Discussion

This chapter has presented statecharts as an executable actor behaviour in Web Prolog, using an SCXML-inspired document syntax and Prolog as both data model and scripting language. The discussion section briefly situates this design relative to prior Prolog–SCXML work, clarifies what the behaviour buys compared to a hand-written `receive/1-2` loop, and then positions the result relative to Erlang/OTP's `gen_statem` and the SCXML Recommendation.

### 6.7.1 Previous work

The idea of using Prolog as a data modeling and scripting language in a statechart formalism is not new. A proof-of-concept implementation of an SCXML system with Prolog as data model and scripting language was developed by a team at the University of Darmstadt in Germany (Radomski et al., 2013). Their focus is multi-modal conversational systems, and the paper is well worth reading, since it argues – contrary to what one might initially expect – that SCXML and Prolog form a good combination for developers of such systems. However, we take a slightly different, and, we believe, simpler and more lightweight approach to the interface between the two languages. Our aim is a profile that harmonizes with Web Prolog and the architecture of the Prolog Web; in other words, although we try to stay as compatible as possible, our goal is to adapt an SCXML-inspired profile to Web Prolog and the Prolog Web, rather than the other way around.

### 6.7.2 Comparison with an ordinary message loop

At this point it is natural to ask why a dedicated statechart behaviour is introduced at all, rather than writing an ordinary message loop in Web Prolog. Small event-driven controllers are easy to code directly, but practical statecharts rely on Harel's extensions such as hierarchy, history, and orthogonality, which buy *structure* that is difficult to recover from ad hoc `receive/1-2` code once the control problem grows. Hierarchy and history give interruption and resumption a first-class representation instead of burying it in auxiliary flags and continuations. Parallel regions support modular decomposition *within* one reactive process, and run-to-completion execution yields deterministic coordination between regions: given the same event in the same configuration, the same transitions fire in the same order. Explicit invocation ties the lifetime of spawned processes to control states, so that leaving a state can reliably terminate what it invoked. Finally, the data model makes long-lived state a declarative Prolog object rather than an opaque tuple threaded through callback code, and the same structure that guides execution is directly visualizable, making control logic easier to review, communicate, and debug.

Beyond these structural benefits, the statechart behaviour also changes the *event-handling posture*. In a hand-written `receive/1-2` loop, selection is driven by the current mailbox contents and by pattern order, while non-matching messages typically remain in the mailbox (effectively deferred). In the statechart profile presented here, event relevance is determined by the current configuration (the active states and regions): one selected event is matched against enabled transitions across the whole configuration, and irrelevant events are discarded unless deferral is modelled explicitly. Conversely, statecharts provide run-to-completion reactions: one external event may trigger a finite cascade of internal work (condition-only transitions, eventless transitions, and internally raised events) before the next external message is consumed. With orthogonal regions, the selected event is visible across the entire active configuration, enabling deterministic broadcast-style coordination within one actor – something that actor-level messaging can emulate only by explicit fan-out and coordination protocols.

Table 6.1 summarizes the most common transition forms in the profile. Seen through the lens of `receive/1-2`, the table can be read as an inventory of control moves that are either awkward or easy to lose track of in an unstructured loop: eventless progress, guarded internal reactions, and targetless event handlers that consume messages without changing state.

A simple rule of thumb follows. A statechart is a good fit when the facets of behaviour are tightly coupled parts of one control process and deterministic, inspectable reactions are desired. Separate actors with ordinary message loops remain the right choice when independent scheduling, failure isolation, distribution, or distinct lifetimes are central concerns.

Another operational difference is that `receive/1-2` is *semi-deterministic*: it either succeeds once or fails. It commits to one selected message (and one matching receive clause); it fails only if the body of that selected clause fails, in which case the failure propagates and may trigger ordinary Prolog backtracking in the surrounding computation. By contrast, the statechart behaviour commits to one external event and then executes a deterministic run-to-completion macrostep under the standard statechart transition-selection rules; executable content is intended to succeed deterministically, with errors routed as internal events rather than expressed via failure.

### 6.7.3 Comparison with Erlang's `gen_statem` behaviour

Erlang's `gen_statem` behavior offers a disciplined way to structure long-running reactive processes: the meaning of an incoming event depends on the current state, and a transition may update state data, schedule timeouts, or emit new events. In that sense, `gen_statem` and our statechart behavior address the same engineering problem: how to express event-driven control in a maintainable way without hand-written receive loops.

| Transition form | Typical use / notes |
|---|---|
| `<go to="s" on="e">...</go>` | Triggered state change. The ordinary event-driven case: on event pattern `e`, transition to `s` and run actions. |
| `<go to="s" on="e" if="c">...</go>` | Triggered and guarded state change. The standard ECA pattern: a matching event plus a condition enables the transition; convenient when several transitions compete on the same event. |
| `<go on="e" if="c">...</go>` | Guarded, targetless event reaction. Consumes events matching `e`, checks `c`, runs actions, and stays in the same configuration. Useful for logging, emitting output, raising internal events, or updating the data model without a state change. |
| `<go to="s">...</go>` | Eventless, unconditional state change. Useful for automatic progress (e.g. completion-style transitions) or compact control-flow steps. Use with care, since chains of eventless transitions may create long run-to-completion cascades. |
| `<go to="s" if="c">...</go>` | Eventless but guarded state change. Useful for decision points driven by the data model, or for encoding automatic progress only when a condition holds. |
| `<go if="c">...</go>` | Guarded, targetless reaction (no control-state change). Useful as a condition-action rule that updates the data model, emits output, raises an internal event, logs, etc., while staying in the same configuration. |

**Table 6.1** Common <go> transition forms and typical uses in the statechart profile.

The statechart behavior differs in where it places complexity. First, it makes the control structure explicit and inspectable as a hierarchy of states rather than as callback code, which improves readability and supports direct visualization. Second, it supports hierarchical states, history, and parallel regions as first-class control constructs, whereas `gen_statem` leaves such structure to be encoded by conventions in the callback logic. Third, the statechart run-to-completion macrostep provides deterministic intra-machine coordination (including broadcast effects across parallel regions under fixed priority rules), while Erlang's coordination across processes is intentionally mailbox-driven and concurrent. Finally, statecharts integrate process invocation as a control construct with a precise lifetime rule (invoked processes are tied to the state that invoked them), whereas `gen_statem` relies on external OTP patterns (supervision, linking, monitoring) to manage process lifetimes.

### 6.7.4 Relationship to SCXML

The textual notation used in this chapter is intentionally SCXML-inspired, but it is not a full SCXML surface language. Our primary goal is to present statecharts as an executable actor behavior in Web Prolog, with Prolog as both data model and executable content. For that reason we aim to implement most of the SCXML 1.0 processing model, while adopting a smaller and more uniform set of constructs that harmonize with actor-style messaging and with Prolog scripting.

Several SCXML constructs are omitted because they are either platform-specific (for example, ECMAScript-oriented scripting facilities), oriented toward browser integration rather than actor systems, or redundant when executable content is restricted to Web Prolog goals. In particular, we do not need a dedicated `<script>` element, since executable content is expressed directly as Prolog code. Likewise, communication with external actors is handled by Web Prolog predicates such as `!/2` (and derived predicates such as `output/1` and `toplevel_call/2-3`), rather than by reproducing the full generality of SCXML's `<send>` element. Moreover, our `<spawn>` element corresponds to SCXML's `<invoke>` but is adapted for actor-style invocation rather than the SCXML service model.

We also rename a small number of constructs for readability in Prolog-centric examples: `<transition>` becomes `<go>`, and the attributes `target`, `event`, and `cond` are rendered as `to`, `on`, and `if`. These are surface changes; the intent is to make examples compact and to emphasize that transitions function as rules governing event-driven evolution of the actor.

### 6.7.5 Toward an actor-oriented successor to SCXML

Seen as a language for programming reactive actors, SCXML's core ideas are compelling, but the Web Prolog setting suggests a slightly different "center of gravity": a declarative data model, a single uniform scripting language and an explicit integration with actor messaging. The statechart behavior in this chapter can therefore be read as a concrete sketch of what an actor-oriented successor might prioritize: the standard statechart execution model, combined with Prolog-level reasoning and with explicit, programmable interaction with the surrounding actor system.

## 6.8 Summary

This chapter introduced the statechart behaviour: an actor behaviour in which control is expressed as a hierarchical, event-driven state machine with parallel regions, deterministic transition selection under run-to-completion execution, and executable actions written in Web Prolog. Statechart actors can be spawned and supervised like ordinary actors, can communicate via messages, and can spawn child actors

with well-defined lifetimes tied to control states. The result is an SCXML-inspired but Prolog-centric profile that supports reactive control, structured coordination within one actor via configuration-wide event visibility, and integration of long-lived knowledge and decision logic through a private declarative Prolog data model.

# Part III
# The broader technological contexts

# Chapter 7
# The Prolog Web ≈ the Semantic Web?

Since the Prolog Web adds logic programming and automated reasoning to the conventional Web, it is natural to ask whether we are approaching something like the *Semantic Web* – the machine-readable Web vision articulated by (Berners-Lee et al., 2001). The gist of their proposal is captured by the following three statements:

> The Semantic Web is an extension of the current web in which information is given well-defined meaning, better enabling computers and people to work in cooperation.
>
> For the semantic web to function, computers must have access to structured collections of information and sets of inference rules that they can use to conduct automated reasoning.
>
> Adding logic to the Web – the means to use rules to make inferences, choose courses of action and answer questions – is the task before the Semantic Web community at the moment.

At first glance, words and phrases such as *logic*, *inference rules*, and *answer questions* could be read as an informal description of what Prolog already does. Likewise, the characterisation as an extension of the current Web fits the Prolog Web rather well. This invites a tempting but slightly misleading conclusion: perhaps the Prolog Web is simply a simpler, "poor man's" Semantic Web?

The resemblance is real, but it is a resemblance of ingredients rather than of engineering goals. The Semantic Web focuses on semantic interoperability by standardising shared meaning – data models, identifiers, vocabularies, and ontology semantics that many independent parties can agree on. The Prolog Web focuses on execution by standardising an executable substrate – a uniform rule language plus a node and process architecture in which predicates can be deployed as running services and agents. Both concern interoperable representations, but the Semantic Web foregrounds representation and shared meaning, whereas the Prolog Web foregrounds turning knowledge into executable behaviour.

The thesis of this chapter is therefore cooperative rather than competitive: the two are best viewed as complementary. In practice, Web Prolog can consume and produce RDF-style data, reuse established vocabularies and identifiers, and apply Prolog's execution model to "put the data to work" as queries, rules, and long-lived agent behaviour.

Although the Semantic Web serves as a convenient reference point, the architectural claims made here apply more broadly to graph-based knowledge systems, including property-graph and hybrid knowledge-graph platforms.

## 7.1 The Semantic Web tower

The Semantic Web is often presented as a layered architecture, where each layer builds on the layers below it. The lower layers are shared with the conventional Web, while the upper layers introduce machine-processable representations of data, query mechanisms, and ontology-level semantics. A commonly used illustration is the so-called *Semantic Web tower*, one variant of which is shown in Figure 7.1.

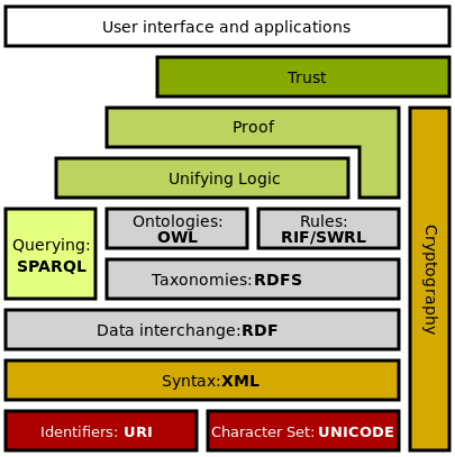



**Fig. 7.1** One variant of the Semantic Web tower.

The tower diagram is necessarily impressionistic, but it provides a useful scaffold for our comparison. At the bottom, URIs play the familiar role of locating resources (as URLs), but on the Semantic Web they also serve as global names for entities, relations and concepts (as IRIs). The Prolog Web uses URIs in this broader naming role too, but it also uses them operationally, for example to locate code and to identify conversations, processes, and even answer streams.

Unicode can likewise be viewed as a naming layer, ensuring that text and identifiers can be represented consistently across languages. Not all Prolog systems fully support Unicode, but a web-facing Prolog system effectively has to.

Many variants of the tower place XML at the syntactic level. In this chapter we treat XML mainly as a data interchange format among several. In the Prolog Web setting, Prolog terms provide a natural internal representation, and JSON is often the most convenient interchange format for client-facing APIs, while RDF serializations

such as RDF/XML and Turtle remain important for interoperability with existing Semantic Web tooling.

## 7.2 RDF as rdf/3

Right above the XML level of the Semantic Web tower we find the levels devoted to semantics proper, where RDF forms the basis. In RDF, triples serve as a universal representation of relations, and through the process of reification the "subject predicate object" (S-P-O) schema can in principle capture all the kinds of relations we might want to represent.

From a Prolog point of view, RDF is a predicate with three arguments (or just two if the functor is used to represent the P in an S-P-O triple). In an ambitious attempt to create a library for working with RDF, Wielemaker et al. (2016) opted for using a predicate `rdf/3`. Here we give an example of triples represented in SWI Prolog taken from the RDF 1.1 primer document:[1]

```
rdf('http://example.org/bob#me',
    'http://...rdf-syntax-ns#type',
    'http://xmlns.com/foaf/0.1/Person').
rdf('http://example.org/bob#me',
    'http://xmlns.com/foaf/0.1/knows',
    'http://example.org/alice#me').
rdf('http://example.org/bob#me',
    'http://schema.org/birthDate',
    literal(type('http://...#date', '1990-07-04'))).
```

Given an RDF database represented in this way, Prolog allows us to define rules in terms of these clauses. For example, to get the birth dates of people somebody knows we can write the following clause in the system created by Wielemaker et al:

```
known_with_birthdate(Me, He, Date):-
    rdf(Me, foaf:knows, He),
    rdf(He, schema:birthDate, Date).
```

SWI-Prolog's RDF libraries support a convenient prefix notation for IRIs. Prefixes can be declared with the library predicate `rdf_register_prefix/2`, after which an IRI may be written in the compact form `Prefix:LocalName`. A similar namespace mechanism would likely be valuable in Web Prolog, both to improve readability and to encourage a disciplined, interoperable naming practice when Prolog code refers to web-scale identifiers.

[1] http://www.w3.org/TR/2014/NOTE-rdf11-primer-20140624/#section-n-triples

## 7.3 SPARQL-style querying in Prolog

Instead of using SPARQL, querying Semantic Web data represented as above can be done in Prolog. The interactive session below illustrates the use of `known_with_birthdate/3` to query the triple store:

```
?- known_with_birthdate(Me, He, Date).
Me = 'http://example.org/bob#me'
He = 'http://example.org/alice#me'
Date = literal(type(xsd:date,'1990-07-04')) ;
Me = 'http://example.org/bob#me'
He = 'http://example.org/john#me'
Date = literal(type(xsd:date,'1992-09-06')) ;
...
```

The key point is modularity. Wielemaker et al. (2016) argue that the Prolog version using `known_with_birthdate/3` has several advantages over the SPARQL version. First, because the Prolog version has an explicit name, it is easy to reuse it in other queries. This allows developers to incrementally build more sophisticated functionality on top of existing, proven and tested building blocks. This can greatly simplify maintenance of complex queries.

Secondly, Prolog is not confined to SPARQL's fixed menu of filters and functions. Any predicate can serve as a guard, transformation, or derived relation at any point in the clause, and the same definition can freely combine RDF triples with other relations – for example, views backed by an RDBMS or calls into domain-specific code.

The query in the above scenario is very simple, but more complex queries are possible too. Control primitives such as `,/2` (conjunction) and `;/2` (disjunction) are guaranteed to work and to have the traditional Prolog semantics, and `\+/1` (NAF) and ISO Prolog aggregation predicates `bagof/3`, `setof/3` and `findall/3`, the de-facto standard library `aggregate`[2] which provides `count`, `sum`, `average`, etc. are available as well.

The SWI-Prolog library `solution_sequences`[3] was developed specifically to allow for a straightforward translation of many SPARQL queries to Prolog and provides `distinct/1-2`, `limit/2`, `order_by/2`, etc. For example, consider the following ten clauses defining a relation `r/2`:

```
r(a, foo).  r(a, bar).  r(a, bar).  r(c, bar).  r(c, bar).
r(b, bar).  r(a, foo).  r(d, baz).  r(a, foo).  r(c, baz).
```

Here is an example of how a SPARQL-like query can be formulated using such predicates, and what the solutions would look like:

---

[2] `https://www.swi-prolog.org/pldoc/man?section=aggregate`

[3] `https://www.swi-prolog.org/pldoc/man?section=solutionsequences`

```
?- limit(10,order_by([desc(C)],aggregate(count,X,r(X,Y),C))).
C = 3, Y = bar ;
C = 2, Y = baz ;
C = 1, Y = foo.
?-
```

Here, solutions are counted *per* value of `Y`, so that on backtracking the call `aggregate(count,X,r(X,Y),C)` enumerates each distinct `Y` for which `r(X,Y)` holds and binds `C` to the corresponding count. The call to `order_by/2` then sorts these `Y-C` pairs by descending `C`, and `limit/2` truncates the result to the top slice – conceptually similar to a SPARQL query with `GROUP BY`, `ORDER BY`, and `LIMIT`.

Of course, we can use a SPARQL-like query such as this one in a call to `toplevel_call/2-3` or `rpc/2-3` as well, although then it may be better to use the `limit` option provided by these predicates instead of the `limit/2` predicate, like so:

```
?- Goal = order_by([desc(C)],aggregate(count,X,r(X,Y),C)),
   rpc('http://n7.org', Goal, [limit(10)]).
```

Note that all this can be handled by an ISOBASE node. However, if we want to inject a program along with the query, we need at least an ISOTOPE node.

The bottom line is that few mainstream general-purpose languages offer an interface to RDF that is as direct and idiomatic as Prolog's: triples map to relations, and rule layers can be expressed in the same language as the queries that consume them.

Wielemaker et al. (2016) summarise the position we adopt:

> We do not consider SPARQL adequate for creating rich semantic web applications. SPARQL often needs additional application logic that is located near the data to provide a task-specific API that drives the user interface. Locating this logic near the data is required to avoid protocol and latency overhead. RDF-based application logic is a perfect match for Prolog and the RDF data is much easier queried through the Prolog RDF libraries than through SPARQL.

This is precisely the role the Prolog Web embraces: Prolog predicates become the task-specific API layer that lives near the data, while remaining compositional, testable, and reusable as ordinary code.

## 7.4 OWL as an external capability

Moving up the Semantic Web tower, we reach OWL, a family of ontology languages based on description logics. OWL's purpose is not merely to provide more rules, but to support a particular kind of interoperability: shared vocabularies with explicit semantic commitments (class hierarchies, property characteristics, domain and range constraints, cardinality restrictions, and so on) that can be processed by standard OWL *reasoners*. This matters whenever independent producers and consumers of

data need to agree not only on identifiers, but on the intended interpretation of those identifiers.

The Semantic Web community makes a clear distinction between knowledge representation languages (such as RDF, RDFS) and query languages (such as SPARQL). As we have seen, Prolog (including Web Prolog) can, to some extent, fill both roles – but with different semantic commitments. Web Prolog is most naturally used as an *executable rule layer*: named predicates encode task-specific queries and derived relations, composable and testable as ordinary code. OWL, by contrast, targets standardisable ontology semantics under open-world assumptions and decidable reasoning fragments, enabling interoperability through shared class and property commitments rather than shared procedural conventions. Prolog therefore excels at turning relational data into reusable executable services; OWL excels at making ontology-level meaning explicit in a form that independent tools and parties can agree on. The Prolog Web is not "an OWL replacement," and should not be presented as one. Web Prolog is grounded in Horn clause logic and, in typical use, relies on closed-world idioms including Negation As Failure. OWL is designed around decidable reasoning in fragments of first-order logic under open-world assumptions, and can express and validate ontology-level constraints in a standard, tool-supported way that plain Horn clauses cannot capture directly without either changing the semantics or moving to richer logic programming extensions.

This does not make the Prolog Web irrelevant to ontology-heavy applications – it merely suggests a division of labour. When OWL is the right tool for defining a shared ontology, Web Prolog can still play at least three valuable roles: *consuming* OWL-derived results (such as classifications from an external reasoner) and combining them with application-specific rules; *mediating* between heterogeneous sources by translating Semantic Web representations into executable relations exposed through the Prolog Web APIs; and providing an *agent substrate* in which long-lived actors use ontology-backed knowledge as one input among many while the agent logic itself remains straightforward Prolog.

Ontologies are thus one of the places where the Semantic Web stack contributes something genuinely distinct: not just inference, but shared semantic commitments and a mature ecosystem of reasoners and tooling. The Prolog Web complements this by making it easier to turn ontology-backed data into running services, executable behaviour, and web-native agents.

## 7.5 Positioning the Prolog Web alongside the Semantic Web

The preceding sections have examined individual layers of the Semantic Web tower and shown how each relates to the Prolog Web. It is now worth stepping back to compare the two ecosystems as wholes. Figure 7.2 places the Prolog Trinity alongside what, by analogy, might be called a "Semantic Trinity": a comparable ecosystem assembled from Semantic Web technologies – RDF and SPARQL for data and

querying, and a general-purpose language such as Java for application logic (and possibly OWL for ontology-level semantics, and an external reasoner for inference).

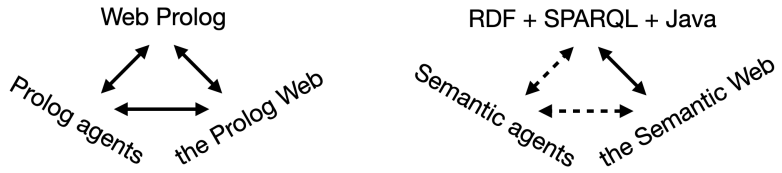



**Fig. 7.2** Comparing the Prolog Trinity with a "Semantic Trinity." Solid arrows indicate tight integration; dashed arrows indicate looser coupling.

Two differences stand out. The first is integration. In the Prolog Trinity the arrows are solid, suggesting that the same language is used for knowledge representation, querying, rule-based computation, and agent scripting, and the execution substrate (nodes, actors, APIs) is part of the proposal rather than an afterthought. The dashed arrows in the Semantic Trinity suggests a less tight integration: the data layer (RDF), the query layer (SPARQL), and the application layer (Java or another host language) are designed and standardised independently, and the glue between them is left to the application developer.

The second difference is architectural. The Semantic Web standards specify representations and query protocols, but they do not prescribe a process infrastructure: there is no standard notion of a Semantic Web node, no standard agent lifecycle, no standard way to host long-lived services that maintain state and coordinate over time. These concerns are left to whatever application framework the developer happens to use. The Prolog Web, by contrast, comes with an architecture as part of the design. Nodes serve answers to queries over standard web protocols, federated querying composes local and remote resources, and code can flow in either direction between client and node. This is not an incidental feature. It is what makes the Prolog Web a *platform* for executable knowledge rather than merely a representation format.

These observations suggest a pragmatic positioning. Semantic Web standards provide interoperable representations and shared semantic commitments, but rich applications still require a host language to implement task-specific APIs, workflows, and integration with heterogeneous data sources. A common assumption is that any general-purpose language will do. In practice, however, there is often an impedance mismatch between graph-shaped, relational data and imperative application code (Wielemaker et al., 2016). This is where Prolog has an unusually good fit. RDF triples map directly to relations, and rule layers can be expressed in the same language as the queries that consume them. Application logic that would otherwise live as opaque client-side code or as ad hoc service glue can instead be expressed as named, reusable predicates close to the data. This echoes the earlier argument by Wielemaker et al. (2016).

From the Prolog Web perspective, then, Web Prolog can act as a programming and agent layer around Semantic Web assets: consuming RDF and SPARQL end-

points when appropriate, delegating ontology-level reasoning to OWL reasoners when needed, and exposing the resulting functionality as web-native services hosted on Prolog Web nodes. The next section makes this complementarity concrete by outlining three integration patterns that recur in practice.

## 7.6 Cooperation patterns

The comparison so far suggests a simple conclusion: the Prolog Web and the Semantic Web are best treated as complementary stacks. The Semantic Web excels at *interoperable knowledge interchange* through shared identifiers, vocabularies and ontology constraints. The Prolog Web excels at *turning knowledge into running services* through executable predicates, a node architecture, and long-lived actors that can coordinate work over time. This section makes that complementarity concrete by outlining three integration patterns that recur in practice, referred to as A, B and C below.

### 7.6.1 A: Web Prolog as a SPARQL client and post-processor

In many applications the most pragmatic role for Web Prolog is to act as a consumer of Semantic Web data offered by existing endpoints. A Web Prolog program issues a SPARQL query to a remote endpoint, parses the result, and then applies ordinary Prolog predicates for filtering, aggregation, explanation, or task-specific inference. The important point is that the *logic that makes the results useful* is written as reusable predicates rather than being forced into a single monolithic SPARQL query.

Conceptually, the pattern looks like this:

1. Query a SPARQL endpoint and obtain a set of bindings.
2. Convert the bindings to a relational representation.
3. Run Prolog rules over that representation and expose the result as a Prolog Web service.

The first step is typically performed by a library predicate (or a small amount of glue code) that implements HTTP and one of the SPARQL result formats. The remaining steps are plain relational programming. If the application is interactive, the derived predicate can be offered through the Prolog Web's APIs, allowing the client to retrieve solutions lazily.

### 7.6.2 B: RDF as `rdf/3`, Prolog rules as the API

A second pattern is to treat RDF as a storage and interchange format, but to treat Prolog as the *programming interface* to that data. In this pattern, RDF triples are ingested into a node-local triple store represented as `rdf/3` facts (possibly backed by an indexed store), while the public surface of the service consists of named Prolog predicates that encode the relevant domain.

The advantage over exposing a SPARQL endpoint is not merely syntactic. Named predicates become stable, reusable building blocks, and they can be composed with arbitrary auxiliary relations, foreign data sources, and task-specific computations. The service remains readable, testable, and evolvable: if a predicate is refactored, callers can remain unchanged as long as the predicate interface is preserved.

In a Prolog Web setting, such predicates can be offered through `rpc/2-3` or through a remote toplevel conversation. The result is a service that is recognisably "Semantic Web" in the sense that it is grounded in RDF identifiers and vocabularies, but "Prolog Web" in the sense that its interface is executable relational code.

#### A minimal bridge example

To make the idea more concrete, suppose a node maintains a local RDF store and provides the above derived predicate `known_with_birthdate/3`. A client can then query this service lazily using `rpc/3`, taking advantage of the Prolog Web's support for bounded slices of answers:

```
?- Goal = known_with_birthdate(Me, He, Date),
   rpc('http://n7.org', Goal, [limit(10)]).
```

The point of the example is not the particular relation, but the division of labour. The node's RDF store and vocabularies ensure that identifiers are interoperable with the Semantic Web, while the exposed predicate provides an application-friendly interface that can be maintained and composed like ordinary Prolog code. If more logic is needed, it can be layered on top of `known_with_birthdate/3` rather than being duplicated across multiple SPARQL queries.

### 7.6.3 C: OWL reasoning as a service, Web Prolog for orchestration

A third pattern is to treat OWL reasoning as an external capability rather than as something to be reimplemented inside Web Prolog. This is often the most realistic approach when an application depends on ontology-level commitments such as class subsumption, property characteristics, and other constraints that are supported by mature OWL reasoners.

In this pattern, OWL plays the role of *semantic backbone*, while Web Prolog plays the role of *executable glue*:

1. An OWL reasoner (local or remote) is used to compute classifications, entailments, or consistency checks.
2. The computed results are imported into Web Prolog as ordinary relations (e.g. `class/2`, `subclass/2`, `entailed/1`, or additional `rdf/3` facts).
3. Prolog rules and, if needed, long-lived actors use these relations as inputs to application-specific decisions, workflows, or interactive services.

This arrangement respects the strengths of both stacks. OWL reasoners provide well-understood semantics and tool support for ontology-level inference, while Web Prolog provides a convenient substrate for integrating that inference with other sources of information, additional rule layers, network calls, and agent-style control logic. The resulting system is typically easier to engineer than an attempt to force all application logic into SPARQL and OWL alone.

### 7.6.4 Summary

The three patterns can be summarised as follows. Pattern A treats Web Prolog as a consumer and post-processor of existing Semantic Web endpoints. Pattern B treats RDF as the data substrate but uses named Prolog predicates as the public interface, thus "hiding" SPARQL behind reusable relations. Pattern C treats OWL reasoning as a specialised service and uses Web Prolog for orchestration and agent behaviour. Together they illustrate the broader theme of this chapter: interoperability and shared vocabularies are the Semantic Web's distinctive contribution, whereas executable relational services and process-level architecture are the Prolog Web's distinctive contribution, and the two therefore compose naturally.

## 7.7 Proof and Trust

The upper layers of the Semantic Web tower are often labelled *Proof* and *Trust*. The labels point to real concerns, but the standards themselves do not prescribe a single, broadly adopted mechanism for either.

For the Prolog Web it is useful to read "trust" in a pragmatic engineering sense. A simple starting point is transparency: a node can let clients inspect the Web Prolog source code behind a published predicate or service, for example through a browsing facility, and optionally accompany it with version identifiers, digital signatures, or provenance metadata. This does not settle trust in any deeper philosophical sense, but it does support auditability, reproducibility, and accountability – which is often what developers actually need.

This stance is also operationally natural in a Prolog setting. A node need not publish only answers; it can also publish the predicate definitions and provenance information required to reconstruct how those answers were obtained. Chapter 8

develops the idea further by showing how proof trees can be collected and inspected as a basis for explainable reasoning in Prolog agents.

## 7.8 User interfaces and applications

The cooperation patterns above are easiest to exploit in the familiar web application architecture: a JavaScript front end plus a back end that exposes task-specific predicates as services.

Semantic Web research has long emphasised the importance of user interfaces (Hachey and Gasevic, 2012). The Prolog Web perspective is pragmatic: conventional Web front ends are typically implemented in HTML, CSS and JavaScript, while Web Prolog services provide a relational and rule-based back end accessed over HTTP or WebSocket. This arrangement is relevant to both approaches discussed in this chapter: a JavaScript UI can talk to Web Prolog services that in turn query RDF stores, SPARQL endpoints, or OWL reasoners, effectively using Web Prolog as an application-logic layer around Semantic Web assets. If Web Prolog is deployed in browsers, a small JavaScript API that exposes its core call and message primitives would likely be the most practical interface.

This is also where agents enter the picture in a concrete, engineering sense. Some agents are *UI agents*: long-lived client-side processes that maintain conversational or task state, trigger background queries, coordinate multiple services, and present results incrementally rather than as a single request–response transaction. In a conventional stack, such behaviour is implemented ad hoc in JavaScript and application glue. In the Prolog Web, the same interaction model can be carried across the client–server boundary: UI agents and back-end services can be written against the same set of message-passing and call abstractions, with the agent logic expressed as stateful actors that communicate by messages, consume RDF/OWL-backed knowledge when relevant, and expose their functionality through web-native APIs. We develop this agent perspective and the disciplined control structures needed to keep it predictable in Chapter 8.

## 7.9 Related work: towers of logic programming

As noted above, many variants of the Semantic Web tower have been proposed, and some align more closely with logic programming than others. The two-tower architecture of Horrocks et al. (2005), reproduced in Figure 7.3, contrasts two plausible instantiations of a layered Web stack.

The right-hand tower captures essentially the same idea as Figure 7.1. The left-hand tower, by contrast, is more explicitly grounded in the logic programming paradigm. Here DLP denotes Description Logic Programs, a fragment designed to connect Datalog-style rule reasoning with part of the ontological expressiveness found in

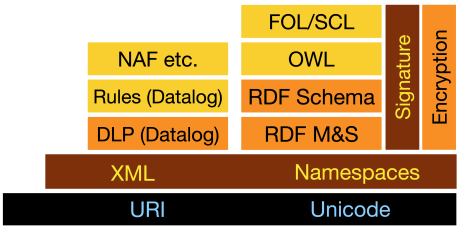



**Fig. 7.3** The two-tower architecture, with the tower proposed in (Horrocks et al., 2005) to the left and a simplified version of the "original" tower to the right.

OWL. Above DLP, one can formulate rules and enrich them with features such as Negation As Failure. The "etc" in the diagram is not mere hand-waving: it points to the broad family of logic programming formalisms surveyed in Section 2.1, each with its own semantic foundations and practical strengths.

The key observation is that this left-hand tower is not a single monolithic stack but a family of possible stacks, each grounded in a different fragment or extension of logic programming. The Prolog Web's profile hierarchy (Chapter 3) provides a natural way to accommodate this diversity. A Datalog engine, with its guaranteed termination and pure relational semantics, is a natural fit for the RELATION or ISOBASE profile. A WFS-capable Prolog system, offering stronger semantic guarantees for negation, can present itself as an ISOTOPE node. A probabilistic reasoning engine can offer a specialised query interface while remaining addressable through standard Web Prolog protocols. In each case, the profile level advertises the system's expressive capabilities, and the protocol layer provides the common ground.

In this reading, the left-hand tower of Horrocks et al. (2005) maps directly onto the architecture described in earlier chapters. The broader vision – a Logic Programming Web rather than merely a Prolog Web – is taken up again in Chapter 11, where we consider what this architecture offers to the wider LP community.

## 7.10 Outlook: Prolog Trinity towers

Where does Web Prolog fit in? One alternative that suggests itself is to allow Web Prolog to form a layer of middleware between code – often consisting of HTML, CSS and JavaScript – that implements user interface and other application code that does not lend itself to implementation in Web Prolog, and the rest. This is suggested in Figure 7.4.

The same kind of arrangement can be built on top of the second tower, as suggested in Figure 7.5.

Or it might be regarded as a *third* tower, independent but interoperable with the other towers. Figure 7.6 suggests this option.

While the Semantic Web might be described as "layered," then, in contrast, the Prolog Web might be described as "integrated" – not only in the sense that the same

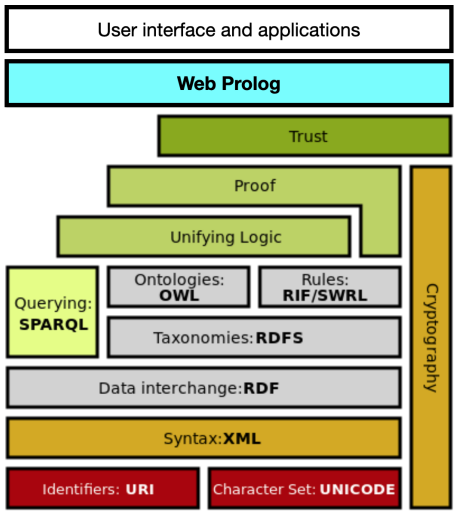



**Fig. 7.4** The Semantic Web tower with an explicit Web Prolog middleware layer.

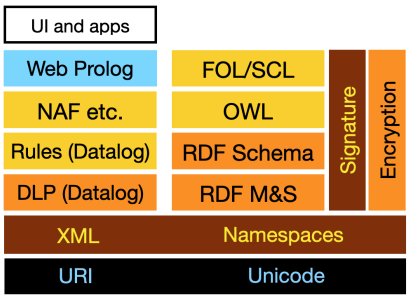



**Fig. 7.5** Two-tower architecture (after (Horrocks et al., 2005)): logic-programming-oriented tower alongside the 'original' tower.

language is used for knowledge representation and querying, but also in the sense that Web Prolog is a full-fledged programming language.

## 7.11 Discussion

The Prolog Web and the Semantic Web are similar in spirit: both treat the Web as a computational environment rather than merely a document store, and both rely on global naming (URIs) and on logic-based machinery to make data more useful than it would be in a purely syntactic web of documents. Both also aim at decentralisation

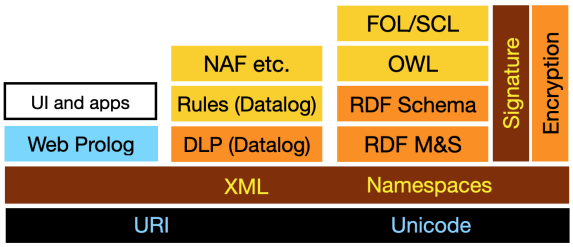



**Fig. 7.6** Three-tower view: adding a Prolog Web tower alongside the two Semantic Web towers (after (Horrocks et al., 2005)).

and federation, in the sense that useful functionality should arise from composing resources offered by many independent parties rather than from a single central database.

At the same time, the two stacks emphasise different things. The Semantic Web places its centre of gravity on interoperable data models and shared semantic commitments, while the Prolog Web places its centre of gravity on executable relational services, process-level structure, and web-native agents. Table 7.1 summarises these differences.

| **The Prolog Web** | **The Semantic Web** |
|---|---|
| Uses the same language for querying and rule-based computation; queries can be packaged as reusable predicates | Typically separates data representation (RDF/OWL) from querying (SPARQL) and application logic |
| Provides a process and service architecture (nodes, APIs, long-lived actors) as part of the proposal | Provides standards for interoperable data and ontology semantics, but does not prescribe a single process architecture |
| Supports agent-style computation naturally through long-lived actors and message passing | Has strong support for shared ontologies and open-world reasoning through standardised languages and reasoners |
| Shared meaning is typically implicit in executable predicates and local conventions; shared ontologies are optional | Interoperability is centred on shared identifiers, vocabularies, and ontology constraints |
| Well suited as a programming and orchestration layer around heterogeneous services (including Semantic Web services) | Well suited as a lingua franca for knowledge interchange across independent parties |

**Table 7.1** Comparing the Prolog Web and the Semantic Web

Although the discussion in this chapter uses the Semantic Web as a concrete point of comparison, most of the underlying issues are not specific to RDF, OWL, or W3C standards. Over the past decade, a range of graph-based technologies – including property-graph systems and large-scale knowledge-graph platforms – have converged on similar concerns: graph-structured data, pattern-based querying, dis-

tributed evaluation, and the tension between shared representation and executable computation. The Prolog Web is intended to address this broader design space, offering an executable logic-based substrate that can sit alongside, or on top of, different graph representations rather than competing with any single one.

It should be clear by now that we do not aim for the Prolog Web to *replace* the Semantic Web, or for Web Prolog to replace the current Semantic Web languages and the technologies surrounding them. On the contrary, we hope and believe that we will be able to carve out a place for the Prolog Web alongside the Semantic Web, and that the languages involved can live happily together. This also appears to be consistent with the multi-stack architecture for the Semantic Web proposed in (Kifer et al., 2005; Horrocks et al., 2005).

Interestingly, Kifer et al. (2005) uses the following analogy when arguing against a single-stack architecture for the Semantic Web:

> While a single-stack architecture would hold aesthetic appeal and simplify interoperability, many workers in the field believe that such architecture is *un-realistic* and *unsustainable*. For one thing, it is presumptuous to assume that any technology will preserve its advantages forever and to require that any new development must be compatible with the old. If this were a law in the music industry (for example) then MP3 players would not be replacing compact disks, compact discs would have never displaced tapes, and we might still be using gramophones.

The authors do have a point, and we would like to draw on the gramophone analogy even further. Quite a few people (usually referring to themselves as "audiophiles") *are* still using gramophones, for the good sound and nice feel of vinyl records. We also note that, for reasons of compatibility and interoperability with more modern technology, the gramophones sold today can usually be plugged into a computer via a USB port. We are thus tempted to think of programs written in Web Prolog as the vinyl records of the logic programming world, and of actors and nodes as the gramophones – updated for compatibility and interoperability with the Web – on which to "play" the programs. And to continue with the analogy, there are things we can do with vinyl records and gramophones, like *scratching*[4] for example, which simply is not possible with newer technologies, and there are things we can do with Prolog, like *programming* for example, which the usual Semantic Web stack of languages does not allow. So while it is true that Prolog is old technology, and while Prolog's tight integration of knowledge representation, querying and general-purpose programming comes with its own set of problems, this integration is at the heart of Prolog. Having it any other way, it would no longer be a Prolog Web.

[4] `https://en.wikipedia.org/wiki/Scratching`

# Chapter 8
# Prolog AI agents on the Agentic Web

The chart in Figure 8.2 describes the evolution of the field of Artificial Intelligence since the late 1950s and up till now. The evolution is described as cyclic, where AI "booms" (or "summers") are followed by AI "busts" (or "winters"). According to this view we have summer now and the excitement over AI has peaked once again and to a height never before seen.

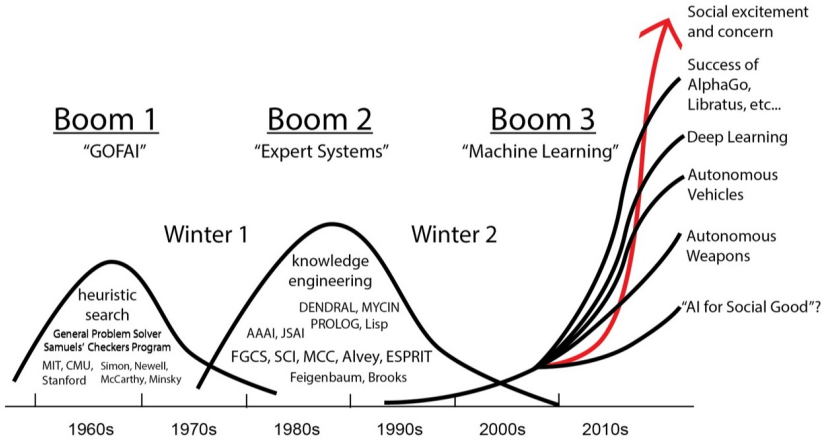



**Fig. 8.1** Cycle of AI booms and busts. Chart inspired by Yutaka Matsuo.

Historically, Prolog and Lisp played prominent roles during the second boom, where the *expert system* was seen as the flagship technology of AI application. As we entered the third boom, the focus shifted toward non-symbolic methods, such as deep learning and Large Language Models (LLMs). However, this chapter argues that the story does not end there. We draw on the rapidly spreading notion of the *Agentic Web*, an evolution of the World Wide Web into an environment where software agents operate alongside traditional resources. While the classic Web serves primarily as a

distributed hypertext system for human consumption, the Agentic Web envisions a dynamic society of autonomous software processes capable of perceiving, reasoning, and acting within the web space.

This vision intersects naturally with logic programming. Prolog's declarative foundations – combined with its suitability for knowledge representation, inference, and dynamic communication – make it an apt candidate for implementing agents that are both knowledgeable and capable of structured interaction. We call this the *Agentic Prolog Web*.

This chapter unfolds in four progressive phases. Section 8.2 establishes the *symbolic competence* of Prolog by revisiting classic AI idioms on a single machine. Section 8.3 *distributes* those idioms across the Agentic Web, transforming predicates into message-driven services. Section 8.5 explores *user-interface* (UI) agents and embodied conversational agents. Finally, Section 8.4 extends this substrate with neural components, showing how Web Prolog can orchestrate Large Language Models to form robust *neuro-symbolic* agents.

===

Figure 8.2 sketches a popular retrospective view of Artificial Intelligence as a sequence of booms and busts: periods of intense optimism and investment followed by "AI winters". On this view, we are currently in a new boom whose scale and public visibility surpass earlier waves.

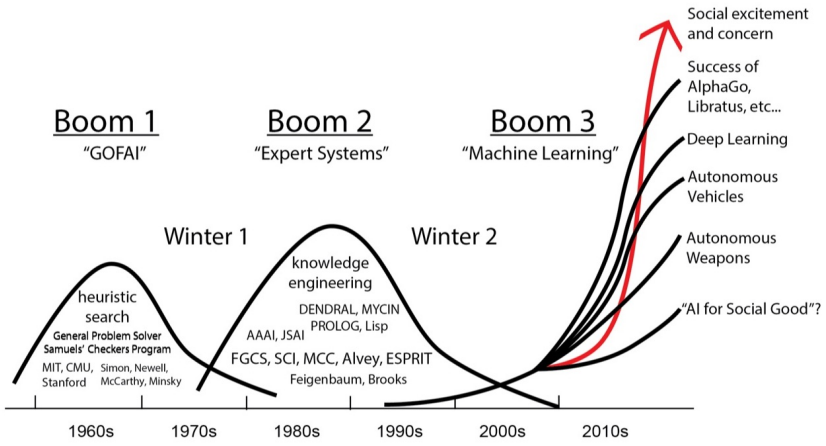



**Fig. 8.2** Cycle of AI booms and busts. Chart inspired by Yutaka Matsuo.

Prolog and Lisp were prominent during earlier waves, especially the expert-systems era, whereas the current boom is driven primarily by data-intensive methods such as deep learning and large language models. This chapter argues that the story does not end there. As software becomes increasingly *agentic*, the need for controllable behaviour and auditable reasoning re-emerges as a first-class engineering concern.

Earlier in the book, and in particular in Chapter 4, we argued that the Prolog Web already constitutes a multi-agent system in a minimal but precise sense: a network of autonomous processes interacting through messages, evolving over time, and embedded in a shared computational environment. This chapter builds directly on that observation. Rather than *discovering* agents as an emergent property of the infrastructure, we now *design* them deliberately. The question is no longer whether agents exist on the Prolog Web, but how Prolog can be used to structure their behaviour, control their interaction, and integrate reasoning, learning, and tool use in a principled and inspectable way.

## 8.1 Framing: why Prolog agents now

The classical Web is primarily a publishing and integration substrate for humans: documents, hyperlinks, and request–response endpoints. A plausible next phase – one that is already taking shape – is a Web populated by long-lived software processes that communicate, maintain state, and take actions over time. The technical heart of the shift is the change in the unit of composition, from pages and endpoints to interacting agents.

The term *agentic* can sound like it merely denotes a new user-interface trend: chatty front ends, tool use, and LLM wrappers. In this chapter we use it in a stricter, architectural sense. An *Agentic Web* is not defined by any particular interaction modality, but by the prevalence of autonomous, communicating processes as first-class web participants. By *Agentic Prolog Web* we mean the fragment of that environment in which agents are implemented as Prolog processes with knowledge bases and reasoning procedures, exposed as web-native services and capable of interacting through message passing.

Historically, the Web has standardised naming, transport, and representation far more than it has standardised *process structure*. We have excellent machinery for addressing and exchanging resources, but comparatively little is fixed about the lifecycle of long-lived services, the discipline of message-oriented interaction, the handling of partial failure, or the way stateful components compose. Early discussions of web agents within the Semantic Web community made this gap explicit: representation and inference mechanisms were advancing, but the process infrastructure in which agents would live and cooperate remained largely an application-level concern (Hendler, 2001).

In practice, this means that agent behaviour tends to accrete in framework glue rather than being captured by a shared, inspectable execution model. The Prolog Web treats this gap as a design target. It keeps the Web's strengths – protocols, naming, interoperability – but adds an explicit execution model: nodes host long-lived actors with mailboxes and supervised lifecycles, and predicates can be exposed as message-driven services. This is why the chapter can move from "agents as an emergent property" to "agents as a deliberate design": the substrate already has the right

shape for autonomous processes, and the remaining problem becomes behavioural engineering – how to specify, constrain, explain, and govern what agents do.

That the architectural gap exists does not, by itself, explain why Prolog should be the language to fill it. Two contemporary pressures make the case.

The first is that agents need enforceable semantics. Large language models have made fluent natural-language interaction cheap and widely available, and they are already becoming an interface layer for many tools. However, fluency is not the same as behavioural reliability. Whenever an agent is expected to respect non-negotiable requirements we need a component whose behaviour has clear meaning and can be validated. Prolog offers precisely this: executable symbolic representations with inspectable derivations, well-defined failure modes, and a clear separation between what is *entailed* and what is merely *suggested*. In other words, Prolog is a natural place to put the parts of an agent that must be auditable, reproducible, and enforceable.

The second pressure is that the modern Web removes the old deployment bottleneck for symbolic AI. Historically, symbolic systems often remained laboratory artifacts: brittle deployments, heavy bespoke infrastructure, and limited interoperability. If an agent had to be installed, configured, and maintained as a special-purpose application, the economic and organisational cost could dominate the technical benefit. Standard protocols, cloud deployment practices, and ubiquitous client platforms change this equation. A Prolog engine can sit behind stable interfaces as a verifier, planner, policy engine, dialogue manager, or explanation service, and it can be composed with other services without losing its semantics.

The resulting positioning is not "Prolog everywhere," but Prolog where semantic clarity and constraint satisfaction matter – what we might call a *behavioural backplane* for agentic systems: the component in which constraints, state, control, and explanations are represented as programs and can therefore supervise more fallible components. This positioning requires candour about Prolog's limitations. A steep learning curve, a fragmented implementation landscape, and limited mainstream ecosystem momentum all constrain adoption. The bet is that these costs are acceptable when the alternative – building constraint enforcement and auditability ad hoc on top of a language that has neither – is worse.

The remainder of this chapter develops the argument in four steps. We begin with a compact reminder of Prolog's symbolic competence on a single machine: knowledge representation, systematic search, planning, meta-interpretation, explanation, and grammar-based language processing. We then lift these idioms onto the Agentic Web by treating Prolog engines as networked actors, so that predicates become message-driven services and proofs can span multiple nodes. Next, we introduce a neuro-symbolic layer in which large language models provide linguistic robustness while Web Prolog supplies the authoritative control and constraint envelope. Finally, we turn to human-facing user-interface agents, where dialogue is best understood as a control problem with states, repairs, and timeouts.

## 8.2 Symbolic competence in one engine

Before we distribute Prolog across the Agentic Web, it is useful to recall what a single Prolog engine already provides as an AI substrate. The point of this section is not to teach Prolog, but to fix a compact capability map: a handful of idioms that repeatedly recur in symbolic AI, and that will later reappear as message-driven services and interacting agents.

### 8.2.1 Knowledge representation and inference

A Prolog program can function simultaneously as a declarative knowledge base and as an executable inference procedure. A canonical example is a small relational theory of kinship, where recursion expresses transitive closure:

```
ancestor_descendant(X, Y) :- parent_child(X, Y).
ancestor_descendant(X, Z) :-
    parent_child(X, Y), ancestor_descendant(Y, Z).

parent_child(X, Y) :- mother_child(X, Y).
parent_child(X, Y) :- father_child(X, Y).

mother_child(trude, sally).

father_child(tom, sally).
father_child(tom, erica).
father_child(joe, tom).
```

A query such as `?-ancestor_descendant(joe,Who)` asks for logical consequences of the clauses and enumerates answers via backtracking. This combination – explicit relational structure plus a proof procedure – is the basic engine behind many symbolic systems, and in Prolog the proof procedure is built in.

### 8.2.2 Combinatorial search: a classic N-Queens program

Many agent tasks reduce to combinatorial exploration: scheduling, placement, configuration, and other problems where one must search a large discrete space under hard constraints. Even in plain ISO-style Prolog, such tasks can often be expressed compactly by representing partial solutions as terms and letting the engine explore alternatives under backtracking, with carefully chosen data representations to prune the search early.

A traditional Prolog solution to the N-Queens puzzle illustrates the style. The program below, due to Richard O'Keefe (1990), encodes the board as a structured set of constraints and constructs a consistent placement row by row:

```
queens(N, Queens) :-
    length(Queens, N),
    board(Queens, Board, 0, N, _, _),
    queens(Board, 0, Queens).

board([], [], N, N, _, _).
board([_|Queens], [Col-Vars|Board], Col0, N, [_|VR], VC) :-
    Col is Col0+1,
    functor(Vars, f, N),
    constraints(N, Vars, VR, VC),
    board(Queens, Board, Col, N, VR, [_|VC]).

constraints(0, _, _, _) :- !.
constraints(N, Row, [R|Rs], [C|Cs]) :-
    arg(N, Row, R-C),
    M is N-1,
    constraints(M, Row, Rs, Cs).

queens([], _, []).
queens([C|Cs], Row0, [Col|Solution]) :-
    Row is Row0+1,
    select(Col-Vars, [C|Cs], Board),
    arg(Row, Vars, Row-Row),
    queens(Board, Row, Solution).
```

The point of including this program in a Web Prolog context is not to claim that a minimal, ISO-oriented profile is the best vehicle for large-scale constraint programming. Rather, the example marks a capability boundary: even without specialised constraint libraries, Prolog can express structured search problems declaratively and solve them by systematic exploration. Where a Prolog system does provide additional facilities such as finite-domain constraints, tabling, or specialised propagators, the same modelling stance can be retained while the engine becomes substantially more efficient. In the Web Prolog profile used throughout this book, however, we restrict ourselves to ISO-level assumptions unless otherwise stated.

### 8.2.3 Planning as search in a state-transition model

Many agent tasks are naturally expressed as planning: find a sequence of actions that transforms an initial state into a goal state. A small meta-program suffices when actions are represented as declarative transition rules:

```
plan(P, P, []) .
plan(P0, Pf, [A|As]) :-
    step(P0, P1, A),
    plan(P1, Pf, As).

step(start, position(monkey, bananas, chair), start).
step(position(Y,Y,Y), final, climb).
step(position(X,Y,X), position(Y,Y,Y), carry(X,Y)).
step(position(X,Y,Z), position(Z,Y,Z), walk(X,Z)).
```

Here the entire world state is a first-order term, and each `step/3` clause specifies an action's preconditions and effects. Prolog's search then discovers action sequences by exploring transition alternatives under backtracking.

### 8.2.4 Meta-interpreters as a control and explanation lever

A distinctive Prolog idiom is to treat programs as data and write interpreters for restricted logics, alternative control regimes, or explanation-producing variants. A minimal meta-interpreter mirrors Prolog's own proof search:

```
prove(true) :- !.
prove((A,B)) :- !, prove(A), prove(B).
prove(A) :- clause(A,B), prove(B).
```

Small changes to such an interpreter can add substantial functionality without changing the object-level knowledge base. For example, adding a second argument allows the interpreter to construct a proof object – a structured explanation that records which rules were used:

```
prove(true, true) :- !.
prove((A,B), (PA,PB)) :- !, prove(A, PA), prove(B, PB).
prove(A, A/Proof) :- clause(A,B), prove(B, Proof).
```

Proof terms of this kind are evidence-carrying data structures that can be inspected, summarised, or shown to a user. In Section 8.3 we will exploit this idiom across nodes, so that a single explanation can span distributed computation.

### 8.2.5 An expert system with a query-the-user facility

Expert systems were a flagship application of symbolic AI during the 1980s. They did not become a lasting mainstream paradigm, but the core pattern remains instructive, especially in Prolog: it shows how deduction can be interleaved with interaction in a disciplined way.

An expert system typically consists of a knowledge base plus an inference engine. A practical system must also be able to acquire facts that are not already stored. In Prolog, a simple way to realise this is to extend the meta-interpreter from Section 8.2.4 with a single additional clause that queries the user whenever a goal is declared `askable`:

```
prove(A) :- askable(A, Q), writeln(Q), read(T), T == yes.
```

The accompanying knowledge base supplies both domain rules and a set of `askable/2` declarations that map predicates to human-readable prompts:

```
good_pet(X) :- bird(X), small(X).
good_pet(X) :- cuddly(X), yellow(X).

bird(X) :- has_feathers(X), tweets(X).

askable(tweets(_), 'Does it tweet?').
askable(small(_), 'Is it small?').
askable(cuddly(_), 'Is it cuddly?').
askable(has_feathers(_), 'Does it have feathers?').
askable(yellow(_), 'Is it yellow?').
```

When `prove/1` encounters an askable subgoal, it suspends deduction, poses the associated question, reads the reply, and then resumes the derivation. For example:

```
?- prove(good_pet(tweety)).
Does it have feathers?
|: yes.
Does it tweet?
|: yes.
Is it small?
|: no.
Is it cuddly?
|: yes.
Is it yellow?
|: yes.
true.
?-
```

The important point is structural: proof search determines which information is needed when. The resulting interaction is mixed-initiative in a precise sense. The engine advances the derivation whenever its knowledge suffices, and it consults the user exactly at the proof obligations that require additional axioms. The `askable` declarations therefore act both as an interaction interface and as a lightweight user model: they mark which facts are expected to come from dialogue rather than from stored knowledge.

This is, of course, a toy shell. In more realistic systems one would add typed answers, richer question forms, explanation facilities, and a persistent dialogue state.

For a fuller treatment of Prolog-based expert systems, see (Merritt, 1989) and (Sterling and Shapiro, 1994).

### 8.2.6 Parsing as deduction: DCGs

Definite Clause Grammars (DCGs) turn parsing into a form of proof search: a sentence is grammatical exactly when it can be derived from the grammar rules. DCGs are also reversible, so the same grammar can be used for generation. Operationally, naive top-down execution makes left recursion problematic, but in tabled Prolog systems this restriction largely disappears: tabling turns parsing into dynamic programming while preserving a declarative rule notation. (Tabling is a system feature rather than part of the ISO core, so it should be read here as an optional acceleration of the same modelling stance.)

### 8.2.7 Summary and handover

The examples above form a compact capability map. A single Prolog engine already supports (i) explicit relational knowledge representation, (ii) systematic search with explicit structure and early pruning, (iii) planning in state-transition models, (iv) meta-level control and explanation via interpreters that construct proof objects, (v) interactive acquisition of missing facts, and (vi) grammar-based language processing as deduction.

The next section lifts these idioms onto the Agentic Web. Instead of treating predicates as internal procedures, we treat them as services: message-driven actors with interfaces, state, and failure modes. The result is an Agentic Prolog Web in which symbolic reasoning is not only executable, but deployable, composable, and inspectable across the network.

## 8.3 Prolog AI agents on the Agentic Prolog Web

The central claim of this section is that the idioms (i)–(vi) do not need to remain local. When Prolog engines are exposed as web-native actors, the same programs can be deployed as distributed services and composed into interacting agents.

The Agentic Prolog Web is therefore not a new symbolic formalism. It is an execution and deployment stance: treat Prolog engines as networked processes with mailboxes, state, and interfaces, and let predicates become message-driven services. In this setting, what used to be a local `read/1` becomes a `prompt/2` message expecting a response; what used to be a local meta-interpreter becomes a mobile

proof-producing component; and what used to be an internal predicate call may become a remote call whose provenance is part of the explanation.

We develop the idea through four ingredients: an expert-system agent that acquires missing facts as a distributed dialogue; a proof-producing meta-interpreter whose explanations span multiple nodes; a positioning of Web Prolog relative to earlier Prolog-based agent frameworks; and a brief discussion of scalability and interoperability on the Web.

### 8.3.1 An expert-system agent and a simulation

Classic expert systems faced two persistent bottlenecks: the difficulty of extracting knowledge from humans and the lack of a ubiquitous infrastructure for deployment. A web-native actor substrate addresses both. By treating the Web as the infrastructure, symbolic agents can be deployed as services; and by treating interaction as message passing, knowledge acquisition becomes an incremental, mixed-initiative dialogue rather than a local read/write loop.

To adapt the askable meta-interpreter pattern from Section 8.2.5 to an agent setting, we replace terminal I/O with an interaction predicate. Concretely, the clause that previously wrote a question and read an answer becomes:

```
prove(A) :- askable(A, Q), input(Q, T), T == yes.
```

The purpose of `input/2` is to decouple deduction from any particular user interface. In a local setting, `input/2` might still be implemented on top of `read/1`. As we saw in Chapter 4, in the Prolog Web it is implemented by sending a `prompt` to some conversational partner and waiting for a reply via `respond/2`.

We can now package the expert system as an expert-system agent by spawning a remote toplevel actor and preparing a call to `prove/1`:

```
expert_ask(Query, Pid, Options) :-
    toplevel_spawn(Pid, Options),
    toplevel_call(Pid, prove(Query), [
        limit(1)
      | Options
    ]).
```

The `limit(1)` option is not an arbitrary restriction. It enforces a simple turn-taking discipline where each remote call either produces a single `prompt` or a single terminal outcome (`success` or `failure`).[1]

To demonstrate the pattern, we construct a small simulation in which a second agent – the shell process in fact – plays the role of the user. The simulation answers

[1] Without such a bound, solution enumeration could interleave multiple answers with interaction steps, complicating the conversational protocol and making it unclear which prompt belongs to which branch of search.

questions stochastically, illustrating that the expert system does not control the dialogue by a script; rather, the proof search itself determines which information must be acquired next.

```
simulation(Options) :-
    random_member(Name, [tweety,pingu]),
    format('A1: Is ~w a good pet?~n', [Name]),
    expert_ask(good_pet(Name), Pid, Options),
    interact(Pid, Name).

interact(Pid, Name) :-
    receive({
        failure(Pid) ->
            format('A2: No, ~w is not a good pet.~n', [Name]);
        success(Pid, _, _) ->
            format('A2: Yes, ~w is a good pet.~n', [Name]);
        prompt(Pid, Question) ->
            format('A2: ~w~n', [Question]),
            random_member(Answer, [yes,no]),
            respond(Pid, Answer),
            format('A1: ~w~n', [Answer]),
            interact(Pid, Name)
    }).
```

One may run the simulation, for example, by spawning the expert-system agent on a remote node, loading the relevant predicates, and providing an initial fact base:

```
?- simulation([
       node('http://n9.org'),
       load_predicates([prove/1]),
       load_list([yellow(tweety)])
   ]).
A1: Is tweety a good pet?
A2: Does it have feathers?
A1: yes
A2: Does it tweet?
A1: yes
A2: Is it small?
A1: no
A2: Is it cuddly?
A1: yes
A2: Yes, tweety is a good pet.
true.
?-
```

The architectural point is that the reasoning process is encapsulated as a mobile, interactive component: an actor whose control flow is governed by logical necessity

rather than by a fixed dialogue script. The Web substrate supplies distribution and composability, while Prolog supplies the proof structure that drives which questions must be asked.

### 8.3.2 Explainable AI across nodes: distributed proof trees

A defining advantage of logic-based, symbolic AI is transparency: every conclusion rests on rules and facts, and an explanation can be constructed as a proof object. On the Agentic Prolog Web, explanations must survive distribution. If a subgoal is proved on another node, its justification should be folded into the same overall proof structure rather than relegated to an opaque remote call.

A compact meta-interpreter can achieve this by treating remote calls as proof-producing sub-derivations. When the interpreter encounters a remote goal, it requests a proof-producing interpreter on the remote node and continues constructing the proof there:

```
prove(true, true) :- !.
prove(rpc(URI,A), Proof) :- !,
    prove(rpc(URI,A,[]), Proof).
prove(rpc(URI,A,Options), Query@URI/Proof) :- !,
    rpc(URI, prove(A, Query/Proof), [
        load_predicates([prove/2])
      | Options
    ]).
prove((A, B), (ProofA, ProofB)) :- !,
    prove(A, ProofA),
    prove(B, ProofB).
prove(A, A/Proof) :-
    clause(A, B),
    prove(B, Proof).
```

The key idea is that remote computation is not treated as a black box. Instead, the same explanation discipline is applied at each node, and remote contributions are marked in the resulting proof term.

Suppose the local node contains the rule `mortal(X) :- human(X)` and also defines `human(X)` by consulting a remote node. If the fact `human(plato)` resides on `n1.org`, we obtain a proof that records the distribution boundary:

```
?- prove(mortal(plato), Proof).
Proof = mortal(plato)/
          (human(plato)/
             (human(plato)@'http://n1.org'/
                true)).
?-
```

Proof-producing interpreters can of course generate very large derivations. On the Prolog Web this is best treated as a controllable engineering choice rather than an unavoidable pathology: one may bound proof depth, abstract away local details and retain only a node-level skeleton, cache recurring subproofs, or compress proof objects into summaries suitable for user-facing explanations. The benefit is that distribution does not force a loss of transparency. Explanations remain first-class data, even when computation spans multiple machines.

### 8.3.3 Meta-interpreters as generic behaviours

The two preceding subsections followed the same architectural pattern. In each case, a meta-interpreter defined the reasoning strategy – askable deduction, proof-term construction – while the actor substrate provided the process container: a mailbox, a lifecycle, and a network address. The domain knowledge was supplied as an ordinary Prolog knowledge base, loaded into the actor at spawn time. The result was a reusable, deployable component whose control flow was determined by logical structure rather than by a hand-written message loop.

This is precisely the separation that generic behaviours are designed to capture. In Erlang/OTP, a generic behaviour such as `gen_server` factors a concurrent service into two parts: a reusable process skeleton (message reception, state threading, supervision hooks) and a set of application-specific callbacks. The programmer supplies the callbacks; the behaviour supplies the process discipline.

Meta-interpreters on the Prolog Web admit the same factoring. The generic part is the interpreter itself – its search strategy, its interaction policy, its proof-construction discipline – together with the actor wrapper that exposes it as a networked service. The application-specific part is the knowledge base: the clauses, the `askable/2` declarations, and any auxiliary predicates loaded at spawn time. Swapping the knowledge base yields a different expert in the same conversational frame; swapping the meta-interpreter yields a different reasoning regime over the same knowledge.

The two examples already presented illustrate the point concretely. The askable interpreter of Section 8.3.1 defines a behaviour in which each proof step may trigger an interactive prompt; the proof-producing interpreter of Section 8.3.2 defines a behaviour in which each proof step extends an explanation term. Both accept the same knowledge-base interface. A third variant could combine the two, producing explanations for askable derivations, by straightforward meta-interpreter composition – a standard technique in the Prolog literature that requires no new infrastructure on the actor side.

What the actor substrate adds is deployment discipline. Each meta-interpreter behaviour, once packaged as an actor, inherits the supervision, addressing, and lifecycle machinery described in the previous chapters. A supervisor can restart a failed proof-producing agent exactly as it would restart any other actor; a client can address an askable agent by URI without knowing which meta-interpreter it runs internally. The generic behaviour boundary thus serves the same purpose on the

Prolog Web as it does in OTP: it localises the reasoning policy inside the behaviour, and exposes only a uniform actor interface to the rest of the system.

The claim here is modest. We are not proposing a fixed catalogue of meta-interpreter behaviours analogous to OTP's behaviours. We are observing that the meta-interpreter tradition already supplies the raw material for such a catalogue, and that the Prolog Web's actor model provides the deployment frame that makes the analogy precise. Whether a richer library of standard behaviours – bounded-resource interpreters, tabled interpreters, interpreters with confidence or provenance tracking – proves useful in practice is an empirical question that the architecture leaves open rather than forecloses.

### 8.3.4 Parsing as remote deduction

A client may offload parsing to a node that hosts a grammar, without embedding that grammar locally:

```
?- rpc('http://n6.org', phrase(s(PT), [joe,saw,a,telescope])).
PT = s(np(pn(joe)), vp(v(saw), np(det(a), n(telescope)))).
?-
```

This vignette illustrates the broader pattern: symbolic competencies that are expensive to ship as libraries or models can be deployed as web-native agents and accessed by clients through uniform remote calls.

### 8.3.5 Implementing agent frameworks in Web Prolog

Logic-based agent languages and frameworks have been implemented in Prolog for decades. Systems such as DALI (Costantini and Tocchio, 2004) and LPS (Kowalski and Sadri, 2015) show that one can retain a Horn-clause core while adding structures for events, actions, reactive rules, and persistent state. Their appeal lies in combining logic programming virtues – declarative clarity, inspectability, and verifiability – with practical mechanisms for autonomy and interaction.

Web Prolog does not present itself as yet another high-level cognitive agent language with built-in planners, ontologies, or a fixed BDI-style architecture. Instead, it provides a lower-level actor substrate in which the primary abstraction is a process with a private database and a mailbox, encapsulated behind a process identifier and supervised by reliability patterns. In this respect it is closer to an Erlang-inspired concurrent logic substrate than to a monolithic agent language.

Precisely because the substrate is low-level and concrete, many classic agent-language ideas can be reconstructed. A receive loop that pattern-matches on incoming messages, together with an explicit representation of pending goals, already suffices to encode reactive rules of the form "on event then action" or "on event and if

condition, then action" as ordinary Prolog clauses over the mailbox and the private database. In statechart actors, this becomes even more structured, allowing rich reactive and mixed-initiative behaviours to be expressed as statechart specifications interpreted by Web Prolog rather than as ad hoc control code.

Seen from this perspective, Web Prolog is again best understood as an enabler rather than a competitor. It supplies the web-native execution fabric on top of which more specialised agent frameworks can be layered. LPS-style engines, DALI-like event-driven agents, and more elaborate frameworks can all be realised as conservative extensions that treat Web Prolog actors as their concrete embodiment on the open Web.

### 8.3.6 Scalability and interoperability on the Web

Two traits of the Web are decisive for symbolic agents: elastic scalability and protocol-level interoperability.

#### Scalability

Web Prolog agents are processes that can be spawned, suspended, or terminated on demand. Because interaction can be mediated by standard web protocols, off-the-shelf deployment practices apply: container orchestration, replication, load balancing, and ordinary observability tooling can be used to scale symbolic services without inventing bespoke infrastructure. Importantly, the actor model supplies natural failure boundaries: one can supervise, restart, and isolate components at the granularity of individual agents.

#### Interoperability

The Web rewards components that speak stable protocols. On the Agentic Prolog Web, terms travel as structured data, which makes it natural to expose symbolic services as typed message interfaces and to interoperate with non-Prolog components through canonical serialisations. Conversely, Prolog agents can call REST services, interact with browser clients, or orchestrate external systems, while keeping their internal reasoning symbolic and inspectable.

Distributed symbolic agents already provide transparency and coordination at Internet scale, but many deployed agents are not purely agent-to-agent components. They are hybrid systems that must incorporate neural components for language, perception, and retrieval, while keeping authority in a checkable substrate. The next section therefore introduces neuro-symbolic agents: architectures in which Web Prolog acts as control and constraint envelope, and LLMs act as pluggable language modules.

## 8.4 Neuro-symbolic AI agents

> To build a robust, knowledge-driven approach to AI we must have the machinery of symbol manipulation in our toolkit. Too much useful knowledge is abstract to proceed without tools that represent and manipulate abstraction, and to date, the only known machinery that can manipulate such abstract knowledge reliably is the apparatus of symbol manipulation.
>
> *Marcus and Davis, 2019*

Large language models are powerful generators of fluent text, and that fluency is precisely why they are not reliability-first systems. When facts, constraints, obligations, or provenance are missing from the prompt, a model can still produce a plausible continuation, and the result may read as confident even when it is wrong. In low-stakes settings this is often acceptable and sometimes useful, but in settings where actions are irreversible or correctness is externally defined (healthcare, compliance, infrastructure automation), we need architectures that can enforce invariants, expose assumptions, and fail safely. Neuro-symbolic agents address this by combining neural components for perception and language with symbolic components for rules, search, and verification, so that generative flexibility is grounded in checkable structure rather than treated as the final authority.

Sections 8.2 and 8.3 established, respectively, the symbolic competence of pure Prolog programs and their web-native, actor-oriented execution model. Building on that foundation, this section shows how neural components (LLMs) and symbolic components (Web Prolog) can be composed into coherent agents, and why Web Prolog is a natural host language for such hybrids when the agent is expected to be concurrent, auditable, and distributed.

The landscape of agentic and neuro-symbolic systems is evolving so quickly that any classification of "current best practice" should be read as provisional. Accordingly, this section focuses on a small set of relatively stable architectural ideas – control (statecharts), constraints (symbolic rules), and supervision (actors) – that remain valuable even as specific tools and model capabilities change.

A further reason to be modest about predictions is that the boundary between description and implementation is itself shifting. It is not implausible that, at some point, one will be able to feed the text of this book – specifications, examples, protocol sketches, and semantic constraints – to a capable LLM and obtain a working implementation of a Prolog node in return. If that happens, the purpose of the architectural material in this chapter is not undermined; rather, it becomes even more important to state the intended semantics clearly, to separate invariants from conventions, and to make the control and safety envelope explicit. Faster code generation increases the value of precise behavioural specifications: it lets us generate implementations more quickly while still knowing what, exactly, they are supposed to implement.

### 8.4.1 Why hybridize?

Neural models excel at perception, linguistic robustness, and learning from heterogeneous data, but they do not natively preserve invariants that a symbolic program can enforce: deductive consistency, traceability, structured explainability, and predictable failure modes. Hybridization therefore aims to combine (i) neural flexibility for perception and language understanding, (ii) symbolic rigour for logic, planning, and verification, and (iii) actor concurrency for scalable orchestration across the Web. The goal is not to replace neural methods, but to embed them in an architecture where the non-negotiables of a domain can be represented and checked continuously.

Table 8.1 contrasts the two paradigms at the level of engineering posture: what kind of semantics they offer, how they fail, and what can (or cannot) be guaranteed. It is meant to clarify why hybrid designs often assign authority to the symbolic layer even when language is handled by an LLM.

A useful way to read the contrast between Prolog and LLM-only systems is in terms of where each one tends to carry engineering weight in deployed agents. Prolog tends to dominate where *correctness is externally defined* and must be enforced rather than narrated. This includes verification and enforcement tasks such as checking constraints, invariants, permissions, and policy, where the system should be able to produce counterexamples or minimal justifications when a requirement is violated. It also includes structured search problems such as planning and combinatorial exploration, where explicit pruning, tabling, or constraint propagation can turn a large discrete space into an executable plan. Finally, Prolog retains a distinctive advantage whenever auditability is a first-class requirement: it can return evidence-carrying artefacts (derivations, failed constraints, proof objects) and support reproducible reasoning steps, and its meaning is comparatively stable over time in the sense that the same program and facts yield the same logical outcomes (modulo control choices).

LLM-only systems, by contrast, tend to dominate where the objective is fundamentally linguistic or heuristic rather than deductive. They excel at fluent generation: drafting text, code sketches, test ideas, documentation, and maintaining interactive dialogue that can tolerate underspecification. They also excel at heuristic exploration: rapidly proposing candidate solutions and creative alternatives without explicit search machinery. Their robustness to messy input is often their most practical advantage: they can accommodate vague intent and heterogeneous knowledge expressed in natural language where a symbolic interface would require careful schema design. Finally, their improvement curve can be unusually steep: capability can jump with new model releases even when the surrounding system and prompts remain unchanged. For hybrid architectures, the design takeaway is not that one side should replace the other, but that *decision authority* is typically placed with the symbolic layer when constraints, audit, and stable meaning matter, while *language capability* and heuristic proposal-making are delegated to the neural layer when the problem is communicative, underspecified, or primarily about drafting and interpretation.

| Posture | Prolog-based reasoning systems | LLM-only systems |
|---|---|---|
| **Semantics and guarantees** | Precise operational meaning; behaviour can be constrained by rules, proof procedures, and resource bounds. | Implicit and statistical; hard to guarantee correctness beyond narrow, externally enforced constraints. |
| **Correctness posture** | Designed to be checked: constraints, invariants, and entailment can be validated and reproduced. | Designed to be plausible: can be accurate, but may also hallucinate or drift without reliable internal validity checks. |
| **Explainability** | Can return structured reasons (derivations, rule firings, failed constraints) and support audit trails. | Explanations are generated narratives; may be persuasive but are not inherently evidence-carrying. |
| **Failure modes** | Typically explicit: non-termination, incompleteness from control choices, or constraint inconsistency; failures can be instrumented. | Often implicit: silent errors, confident wrong answers, instruction-following failures, and sensitivity to prompt phrasing. |
| **Data and learning** | Knowledge is explicit and editable; learning is external (rule authoring, induction, or updates to facts/rules). | Knowledge is latent in parameters; updating behaviour often requires retraining, fine-tuning, or additional scaffolding. |
| **Integration shape** | Natural fit as a verifier, planner, policy engine, or reasoning service; can sit behind APIs with stable semantics. | Natural fit as a generator and interface layer; needs surrounding machinery for validation, safety, and state consistency. |
| **Strength under constraints** | Strong when requirements must be satisfied (safety, compliance, configuration, scheduling). | Strong when requirements are soft and language-driven (drafting, summarizing, proposing options). |
| **Cost model** | Predictable per inference step; performance depends on search space and control but can often be bounded. | Token- and model-size-driven; costs can be high and scale with context length; behaviour may vary across model versions. |

**Table 8.1** Contrasting Prolog-based reasoning systems with LLM-only systems: engineering posture and guarantees.

### 8.4.2 What is neuro-symbolic AI?

A neuro-symbolic system is any computational architecture in which neural and symbolic modules exchange intermediate representations at run time under the control of an agent framework. The coupling may be loose (black-box tool calls with structured I/O), semi-tight (shared constraints, schemas, or retrieval paired with symbolic checks), or tight (a single differentiable pipeline in which parts of the symbolic procedure are compiled into a learnable computation graph). These three modes correspond to three engineering postures: orchestration, co-representation, and end-to-end training.

### 8.4.3 Architectural patterns for hybrid agents

Four patterns recur in contemporary neuro-symbolic agent design.

#### Reflection

An LLM invokes external tools for objective sub-tasks such as retrieval, arithmetic, symbolic query answering, or constraint checking. Instead of fabricating a result, the neural component delegates, and the returned value constrains the next generation step. In practice, this is the simplest way to reduce factual drift and to obtain verifiable intermediate results.

#### Planning with verification

The LLM proposes an explicit action sequence while the symbolic layer checks preconditions, postconditions, and resource constraints before execution. The LLM is treated as a planner and summarizer, while Web Prolog plays the role of verifier and dispatcher. This pattern is especially useful when actions are partially ordered, when actions can fail, or when the environment imposes hard constraints that must never be violated.

#### Multi-agent decomposition

Sub-goals are assigned to cooperating actors so that each specialist works locally while the overall system remains concurrent and robust to partial failure. The neural component may participate as one actor among others, or it may be called by several actors as a shared service. In both cases, actor boundaries provide a natural granularity for supervision, timeouts, and audit trails.

#### Finite-state or statechart cage

The LLM is embedded inside a deterministic control wrapper that constrains what can happen next. The wrapper can accept, reject, or repair model outputs, and it can route the dialogue through explicit states such as `collect`, `confirm`, and `abort`. In Web Prolog this wrapper can be implemented directly as a statechart actor, giving the system a concrete behavioural skeleton rather than an implicit, prompt-only one.

A closely related engineering stance is articulated by Adam Charlson in his 2025 talk on building multi-agent systems with finite state machines. There, finite state machines (more precisely, statecharts) are presented as a governance layer around LLM-driven systems. The core idea matches the cage pattern used in this chapter: the LLM may propose actions or emit events, but an external statechart

remains authoritative, enforcing valid transitions via guards and refusing to act on unmodelled events (which are instead logged for remediation). In this sense, state machines supply predictability, observability, and control that unconstrained text-generation loops typically lack, while the LLM contributes flexible language and task-specific heuristics inside the supervised control envelope.[2]

### 8.4.4 Web Prolog as symbolic backend and agent layer

Python dominates ML practice for good reasons, and the point of this chapter is not to argue against Python. The point is that a hybrid agent needs more than tensor libraries: it needs an orchestration substrate with first-class support for search, symbolic constraints, concurrency, supervision, and network-transparent messaging. Web Prolog places these capabilities in the language model itself.

Web Prolog offers built-in backtracking and nondeterminism, which lets an agent represent alternatives and explore them under logical control rather than by ad hoc sampling. It supports guarded message reception and mailbox-driven coordination, which makes it natural to express reactive, concurrent control flows without collapsing everything into a single event loop. Its actor primitives and supervision patterns provide a principled way to handle partial failure, retries, and escalation. Finally, because messaging over the Web is part of the language-level model, the boundary between local and remote computation can be treated uniformly, which matters when a hybrid agent is distributed across nodes and services.

The result is a single substrate in which symbolic reasoning, orchestration, supervision, and network transparency are expressed in one semantics. Neural components then become pluggable co-processors rather than the place where control logic silently accumulates.

#### Normative reasoning as executable policy

Legal, ethical, and organisational constraints can be modelled as rule sets in Prolog, including defeasible rules, exceptions, and priority schemes. This makes it possible to treat compliance not as post-hoc documentation but as executable policy: before an action is taken, the agent can attempt to prove that it is permitted under the current facts, and if no proof exists it can either refuse, ask for missing information, or escalate to a human.

Even toy examples such as Asimov's laws are useful here, not because they are realistic, but because they illustrate a general point: when constraints are represented as rules, they can be queried, tested, audited, and monitored at run time. For computational-law treatments in a Prolog setting, see, for example, the discussion

[2] This talk is available at `https://youtu.be/OD13PiXw60o`.

in *Prolog: The Next 50 Years* and related work on executable norms and obligations (**??**).

#### Bridging, not siloing

Hybrid architectures are not an either/or choice between Prolog and Python. Modern Prolog systems increasingly support tight, bidirectional interoperation with Python, and Janus in particular provides a fast two-way interface between Prolog and Python that is being developed as a de facto standard across multiple Prolog implementations. From an agent-design perspective, a common and often natural direction is for Prolog to call Python: the symbolic layer retains control and delegates vectorised numerics, deep-learning inference, or GPU-backed tensor algebra to Python libraries, while Prolog remains responsible for search, constraint enforcement, supervision, and auditability. This yields a pragmatic division of labour: Python for tensors, Prolog for control (**??**).

### 8.4.5 Interaction patterns between LLMs and Prolog

This subsection describes three progressively stronger integration styles. They are not mutually exclusive, and real systems often combine all three.

#### LLM → Prolog (tool invocation)

In a tool-augmented architecture, an LLM is embedded in an orchestrator that can route structured requests to external components. If we expose a Prolog engine as a tool, the LLM can decide when to delegate a subproblem to symbolic evaluation, and the result becomes a grounded intermediate value rather than a fabricated continuation. The essential design point is that the Prolog side must be sandboxed: allowed predicates are whitelisted, resource limits are enforced, and the result is returned in a typed, machine-checkable format.

Conceptually, one may imagine a helper predicate such as `prolog_tool/2` that evaluates a restricted query and returns a structured answer term that the LLM can incorporate into its next step. This is the lightest-weight integration, and it already delivers substantial benefit when the delegated subtask has crisp semantics.

#### Prolog → LLM (API delegation)

Conversely, Web Prolog can delegate open-ended language tasks to an LLM and treat the model call as just another effectful predicate, guarded by timeouts, budgets, and policy checks. The exact API surface will evolve, but the architectural point

remains stable: the symbolic layer controls when to call the model, how to constrain the prompt, how to validate the reply, and what to do when the call fails or returns something unsafe.

The following sketch shows the shape of an HTTP call to a contemporary text-generation endpoint. The details should be read as schematic and should be aligned with the provider's current documentation. Note that the option access is expressed using `options/3` rather than field access syntax.

```
:- use_module(library(http/http_client)).
:- use_module(library(http/http_json)).

call_llm(SystemMsg, UserMsg, Options, Reply) :-
    options(Options, endpoint, Endpoint),
    options(Options, authorization, Auth),
    options(Options, model, Model),
    Payload = json([
        model=Model,
        input=[
            json([role=system, content=SystemMsg]),
            json([role=user,   content=UserMsg])
        ]
    ]),
    http_post(Endpoint, json(Payload), Reply,
              [ request_header('Authorization'=Auth)
              ]).
```

In a Web Prolog setting, this call is typically wrapped in a control shell: a timeout guard, a retry policy, a trace hook that records prompts and replies with provenance tags, and a validator that checks the reply against a schema before it re-enters the agent's control flow.

### Tight coupling and differentiable logic

In tight neuro-symbolic systems, parts of the symbolic procedure participate directly in gradient-based training. The core idea is to replace crisp truth values with differentiable scores, and to compile proof search into a computation graph that can be optimized. Projects such as DeepProbLog and NeurASP demonstrate that this can be engineered in practice by compiling logic programs into weighted proof graphs or tensor expressions executed in a deep-learning framework (**??**).

From the Trinity perspective, the key observation is not that every agent should be differentiable, but that Prolog already provides a disciplined representation of structure, and that structure can sometimes be made learnable where it is genuinely useful, for example when learning soft constraints, priors, or perception modules. A Web Prolog node can still keep behavioural authority in explicit rules and policies, while allowing selected components to be optimized statistically.

### 8.4.6 Statechart-wrapped medical triage agent

To make the cage pattern concrete, consider a simplified medical triage agent implemented as a statechart actor supervising an LLM.

1. A Web Prolog actor initialises a statechart with states `collect`, `confirm`, `diagnose`, `advise`, and `abort`.
2. In `collect`, the LLM generates patient-facing questions; answers are parsed by DCGs and stored as structured facts.
3. The transition `collect` $\rightarrow$ `confirm` fires when required slots such as `age/1` and `symptom/1` are instantiated.
4. In `diagnose`, Prolog rules map symptom sets to candidate conditions and compute risk scores under explicit assumptions.
5. If risk $\geq \theta$, the statechart transitions to `advise(escalate)`; otherwise to `advise(self_care)`.
6. All model outputs pass through a validator before they re-enter the state machine, rejecting disallowed content (for example, prohibited advice classes, missing disclaimers, or policy violations) and forcing the agent into `abort` or `escalate` when appropriate.

The central benefit is architectural: the LLM provides language, but the statechart provides behaviour. The system therefore has states, transitions, stopping conditions, and escalation paths, which are difficult to guarantee in a free-form prompt loop.

### 8.4.7 Challenges and research directions

- **Latency and budgeting.** When LLM calls sit on a critical interaction path, the agent must treat time and cost as first-class resources, using caching, local models, or co-location where appropriate, and degrading gracefully when budgets are exceeded.
- **Tracing and audit.** Bridging symbolic traces (choice points, proof trees, constraint stores) with neural traces (prompts, tool calls, sampling settings) into a unified audit log remains a practical challenge, but it is also a major opportunity for web-native agent engineering.
- **Proof artefacts as user-facing explanations.** Proof trees and derivation traces are precise but not always pleasant to read. An LLM could be used as a commentary layer that turns such traces into structured explanations (step-by-step narratives, minimal justifications, or counterfactual "what would have to change" analyses), while the Prolog artefact remains the authoritative source of truth.
- **Intermediate representations.** Standardising typed schemas for the boundary between neural and symbolic modules reduces brittle prompt engineering. JSON with explicit types is often sufficient, but richer formats such as JSON-LD can help when provenance and linking matter.

- **Safety envelopes.** A reliable agent needs a policy layer that is external to the model and enforceable by construction. In Web Prolog this naturally lives in the symbolic backplane and in the supervisory wrapper.
- **Differentiability where it matters.** End-to-end gradient flow through parts of a logic program remains an active research direction, and it is most compelling when used selectively rather than ideologically.
- **Traffic optimisation on the Prolog Web.** Because Web Prolog makes messaging and distribution explicit, it naturally exposes telemetry (latency, queue depths, error rates) that could support learning-based optimisation of higher-level traffic patterns on the Prolog Web, such as pacing, batching, replica selection, or load-aware routing. The architectural twist is that this can be done with hard safety and policy constraints enforced by the symbolic and actor-level substrate: learning chooses among allowed actions, while Prolog defines what is allowed (**??**).

### 8.4.8 Summary: Toward a web-native neuro-symbolic MAS

By placing Web Prolog at the heart of the agent architecture, we inherit constraints, symbolic search, actor-oriented concurrency, and supervision from the outset. LLMs then become pluggable neural co-processors rather than monolithic sources of truth. Compared with Python-centric stacks that treat control as an emergent property of prompts and glue code, the Web Prolog approach makes control a program. This is the core architectural claim of this section: neuro-symbolic agents are not merely LLMs with add-ons, but systems whose behaviour is defined by a symbolic substrate that can call neural components when language or perception is needed.

Hybrid architectures still need to communicate their state, assumptions, and constraints to humans. The most common place where these choices become user-visible is dialogue: turn-taking, repair, refusals, and escalation are precisely where a symbolic control envelope pays off. We therefore turn next to user-interface agents.

## 8.5 User-interface agents

Content is king. – *Bill Gates*

Conversation is king. Content is just something to talk about. – *Cory Doctorow*

The Agentic Prolog Web is a substrate for multi-agent interaction in general, but many of the most common and economically important agents are not peers in a negotiation protocol. They are user-interface agents: components that mediate between a person

and a computational environment. In such settings, the engineering problem is not only what the agent can infer, but how it manages an interaction over time: when to ask, when to confirm, how to repair misunderstandings, how to handle silence and timeouts, and when to refuse or escalate.

This section therefore shifts emphasis from agent-to-agent coordination to dialogue as control. The central claim is that conversational behaviour should not be treated as an emergent property of a prompt. It should be treated as a program with states, transitions, and failure handling. Web Prolog is well suited to this stance because it can combine (i) symbolic representations of dialogue-relevant facts, (ii) executable parsing and interpretation, and (iii) an actor-oriented control shell, most naturally expressed as a statechart.

### 8.5.1 UI agents versus MAS agents

In classic multi-agent systems, interaction is often modelled as peer-to-peer exchange: negotiation, coordination, and distributed problem solving among relatively symmetric participants. User-interface agents are asymmetric. They exist to serve a person, and they are evaluated by human-facing criteria such as clarity, trust, timing, politeness, and error recovery. The success condition is not only that the agent eventually produces a correct plan, but that it maintains an intelligible and safe interaction while doing so.

On the Web, this role frequently manifests as a conversational web agent: an agent that accepts input in natural language (text or speech) and produces responses that guide the user through a task. In such systems, the boundary between reasoning and interaction is not clean. Questions, confirmations, clarifications, and refusals are themselves action choices that must be governed by policy.

### 8.5.2 Turn-taking and repair as control structure

Conversation, in the technical sense used in linguistics, is a jointly managed activity. Participants alternate turns under a largely implicit turn-taking system; many contributions occur in adjacency-pair patterns (question–answer, offer–acceptance, request–grant); and when misunderstanding occurs, participants engage in repair sequences to re-establish common ground.

For an engineered UI agent, the practical implication is that interaction cannot be reduced to a single request–response call. The agent needs persistent state and control over what may happen next. It must be able to suspend a task to ask a question, resume after an answer, back up after a misunderstanding, and time out or escalate when interaction fails. These are precisely the kinds of behaviours that state machines and statecharts are designed to express.

A useful mental model is to separate the conversational system into a *flow layer* and a *content layer*. The flow layer governs when the system may advance, when it must confirm, what to do on silence, and how to recover after a misunderstanding. The content layer is where parsing, belief update, and reasoning live. In a Web Prolog setting, both layers can be expressed as programs, but the flow layer benefits from an explicit statechart representation because the control surface is the point where reliability and user trust are won or lost.

### 8.5.3 Dialogue as proof: the missing-axioms view

A useful way to connect dialogue management to symbolic AI is the Missing Axioms Theory (Smith & Hipp, 1994). In this view, the agent's task can be phrased as an attempt to prove a goal. When the proof fails, the reason is often not that the rules are wrong, but that some required facts are missing. Dialogue then becomes a controlled process for acquiring the missing axioms.

This perspective is already latent in the askable expert-system pattern from Section 8.2.5 (and, in web-native form, Section 8.3.1). There, proof search determines which question must be asked next. The limitation of the toy meta-interpreter approach is not the idea, but the control envelope: real dialogue involves interruptions, repairs, topic shifts, and timeouts. To handle those robustly, we need an interaction controller with persistent state and well-defined failure handling.

### 8.5.4 Statecharts as dialogue governors

A practical architecture is to separate flow from content. The flow concerns turn-taking, timeouts, confirmations, and repair. The content concerns parsing, storing extracted facts, and reasoning about constraints and next steps. Statecharts are a good fit for the flow layer because they support hierarchy and orthogonality, events, guarded transitions, actions, and a history mechanism.

In Web Prolog, a statechart actor can therefore serve as the authoritative dialogue governor, while Prolog code in the statechart's data model provides the symbolic substrate for interpretation and reasoning. A minimal illustration is an agent that collects two attributes (colour and size) and uses a history state to recover after a misunderstanding. Operationally, this means that the dialogue manager needs a representation of where it is in the interaction, what information is still missing, and what repair strategies are permitted.

```
<statechart datamodel="web-prolog" initial="Start">
  <datamodel>
    :- dynamic color/1, size/1.
    np(Attr) --> [a], adj(Attr), [apple].
    adj(color(red)) --> [red].
```

```
    adj(size(big))  --> [big].
  </datamodel>

  <state id="Start" initial="Color">
    <history id="H">
      <go to="Color"/>
    </history>
    <state id="Color">
      <onentry>
        writeln('What color do you want?')
      </onentry>
      <go to="Size"
          on="input(Input)"
          if="phrase(np(color(C)), Input)">
        assert(color(C))
      </go>
    </state>
    <state id="Size">
      <onentry>
        writeln('What size do you want?')
      </onentry>
      <go to="Final"
          on="input(Input)"
          if="phrase(np(size(S)), Input)">
        assert(size(S))
      </go>
    </state>
    <go to="H" on="input(_)">
      writeln('I did not understand.')
    </go>
  </state>
  <final id="Final">
    <onentry>
      size(S), color(C),
      format('You want a ~p ~p apple!~n', [S,C])
    </onentry>
  </final>
</statechart>
```

Three elements matter here.

### Repair via history

The history state supports a simple repair policy for unrecognised input. When an `input(_)` event fails to match any more specific transition, the machine emits a

complaint and transitions to the history pseudostate. This re-enters the most recently active substate of the dialogue, causing its `<onentry>` prompt to run again and thereby re-asking the current question. In contrast to restarting the whole interaction, the repair stays within the current plan, resembling how human repair sequences typically preserve the larger conversational trajectory.

#### Parsing as a guard

Transition guards call DCG parsing directly, so the controller advances only when the incoming `input(Input)` can be parsed into the intended structured form. This makes the interface between interpretation and control explicit: the grammar determines what counts as understood, while the statechart determines when understanding is sufficient to commit to a control move.

#### Belief updates as explicit effects

Executable content makes belief updates explicit and localised. When an input is successfully parsed, the extracted fact is asserted as a named predicate in the statechart's data model. These stored facts can later be queried, audited, and constrained by ordinary Prolog rules, rather than remaining implicit in conversational context.

Together, history-based repair, guard-based interpretation, and explicit data-model updates yield a compact pattern for dialogue control: recognise → commit → store, otherwise complain → repeat.

### 8.5.5 Handover: dialogue as a governed interface

Statechart-governed dialogue provides predictability, repair structure, stopping conditions, and a principled place to implement refusals and escalation. In the neuro-symbolic setting developed in Section 8.4, this control envelope plays an additional role: it defines the boundary between what the language model may suggest and what the agent may do.

Seen this way, the statechart is not merely a dialogue manager. It is a governor for an LLM-augmented interface: the model can be invoked for paraphrasing, interpretation, and question generation, but its outputs re-enter the system only through guards, schemas, and policies that are authoritative and auditable. Dialogue thus becomes the most visible instance of the general design stance of this chapter: place behavioural authority in the symbolic substrate, and use neural components as pluggable co-processors where language robustness is needed.

## 8.6 Summary

This chapter framed Prolog's relevance in an AI-driven software landscape as an operational question: can we make symbolic constraints, search, and explanations deployable and composable at Internet scale, so that they can supervise more fallible components in agentic systems. We first recalled Prolog's symbolic competence on a single machine: representing knowledge as executable clauses, exploring discrete spaces systematically, planning as search, constructing proof objects for explanation, acquiring missing axioms through interaction, and using DCGs to connect language and structure. We then lifted these idioms onto the Agentic Web, treating Prolog engines as networked actors and letting proofs and explanations span nodes.

On that foundation, we argued for a neuro-symbolic stance: treat large language models as powerful but fallible language modules, embed them in a symbolic control and constraint envelope, and make behavioural authority explicit. We noted that the ecosystem changes rapidly, and that even code generation itself may become increasingly automated, but that this makes precise semantics and explicit invariants more valuable rather than less.

Finally, we turned to user-interface agents as the most common and user-visible application class of these ideas. Dialogue is where control matters: turn-taking, repair, confirmation, timeouts, refusals, and escalation must be governed by something more concrete than a prompt. Statecharts provide this governance, and in Web Prolog they can be executed as actors whose guards and effects are symbolic programs. In that setting, the LLM becomes a co-processor for language, while the statechart and the Prolog substrate remain responsible for what the agent is allowed to do, what it knows, and why it believes what it believes.

# Part IV
# From architecture to paradigm

# Chapter 9
# The Trinity as a web-native symbolic paradigm

The preceding chapters have introduced the elements of the Prolog Trinity ecosystem. We began with Web Prolog as a carefully delimited profile of Prolog, extended with Erlang-style message passing and concurrency-oriented semantics. We then examined Prolog agents – actors and nodes – as concrete computational entities, each equipped with clearly defined capabilities and lifecycle properties. From there, we explored the Prolog Web itself: the network that arises when such entities are allowed to interact across transport boundaries, using both sequential query-answer patterns and long-lived concurrent conversations. Subsequent chapters introduced reusable behavioral patterns, statechart actors, and the positioning of the Trinity in relation to the Semantic Web and contemporary AI architectures.

Taken individually, these components may appear as incremental extensions of existing technologies. Web Prolog resembles Prolog with additional concurrency primitives. The actor abstraction draws inspiration from Erlang. The sequential layer echoes long-established patterns of remote querying. Statecharts are extensions of ordinary event-driven state machines.

Yet when considered together, these elements form a coherent architectural stance – one that may reasonably be viewed as a candidate paradigm for executable symbolic systems on the Web. And perhaps not *only* symbolic. Neuro-symbolic integration is a growing area across the AI community. The Web is becoming agentic, and it seems evident that a language that can serve both as a logic programming language and as an agent programming language may have something to contribute.

With that framing in place, this chapter asks a disciplined question:

> What conceptual pattern emerges when logic programming, actor-based concurrency, Web transport semantics, and structured behavioral abstractions are deliberately integrated into a single design?

The answer, we will argue, is a paradigm characterised by a small number of interlocking commitments: a foundational separation between sequential and concurrent computation; a notion of portability grounded in interaction rather than implementation agreement; the agent as the primary unit of structure; a logical foundation

that survives distribution; and the treatment of interaction governance as a first-class design dimension.

These commitments are mutually constraining. The language profile determines what can be exposed safely and portably; the agent model determines which long-lived behaviours can be made robust; the network substrate determines which forms of composition are realistic at Web scale. It is precisely this mutual constraint that prevents the proposal from collapsing into "Prolog over HTTP", "actors in Prolog", or "a distributed Prolog database". No single strand can be redesigned in isolation without consequences for the others.

## 9.1 The two partitions as a foundational principle

One of the central structural decisions in the Trinity architecture is the separation between two distinct partitions of computation: the sequential and the concurrent.

The *sequential partition* is the world of queries and relations. A client poses a query, a node evaluates it against a clause database, and solutions are produced nondeterministically. Over the Web, this partition manifests as remote querying, paged solution retrieval, and the disciplined mapping of backtracking onto HTTP-level interactions. The programmer's mental model remains close to that of classical Prolog, even though transport boundaries intervene.

The *concurrent partition* is the world of actors and interactions. Computation is structured around long-lived processes equipped with mailboxes, identities, and lifecycle semantics. Communication is explicit, asynchronous by default, and potentially distributed across nodes. Communication-failures and recovery are first-class design concerns.

In the Trinity, the two partitions are deliberately kept orthogonal at the programming model level. A sequential query may be realised as an actor conversation underneath, but the programmer need not know or care; conversely, an actor managing a long-lived protocol is not forced to reason about backtracking and choice points. Each mode has its own transport assumptions, blocking surfaces, and communication-failure semantics.

This separation provides a stable design axis. Systems can be built primarily in one partition while selectively invoking the other. Sequential queries may spawn concurrent actors. Actors may internally perform sequential reasoning. Yet the boundary remains conceptually intact.

Beyond the partition boundary, several orthogonal axes further structure the design space. The most important is the axis of *purity*. If the remote goal in an `rpc/2-3` call is pure, the call itself is pure; answers are logical consequences of distributed knowledge bases. The impure Prolog Web incorporates procedural and extra-logical features – cut, dynamic database updates, I/O – as well as the inherently procedural character of actor-based concurrency. A related axis is *openness versus risk*: the extent to which a node exposes itself as a shared execution environment, with the attendant security and resource-governance pressures. The profile hierarchy and the

capability posture developed in earlier chapters are best read as disciplined responses to both axes.

In practice, systems mix pure and impure freely: pure queries for declarative reasoning, actors for coordination and lifecycle management. The guiding principle is *incremental relaxation*: begin with the purest model that fits the problem, relax purity only when necessary, and prefer explicit structure over implicit behaviour. Section 9.4 develops the purity axis in full.

### 9.1.1 Two levels of abstraction

The sequential partition gives rise to a striking property: the Prolog Web can be read simultaneously at two levels of abstraction. On the upper level lies a *level of logic programs* – interlinked Prolog clauses distributed across nodes, where URIs serve as references between knowledge bases and `rpc/2-3` calls serve as the operational links. On the lower level lies a *level of communicating processes* – often realised as toplevel actors that are created, exchange messages, and eventually terminate.

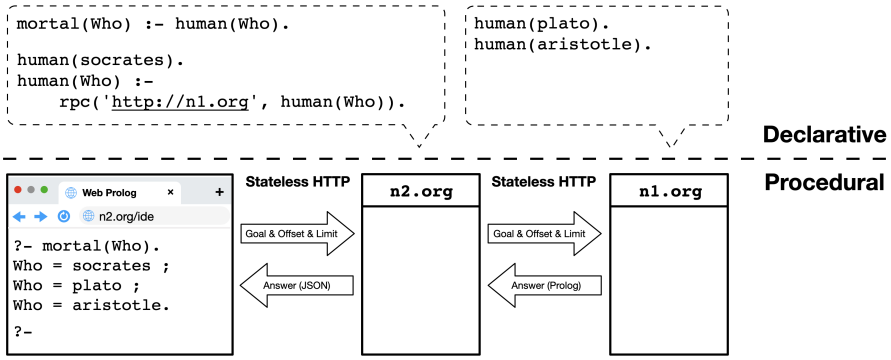



**Fig. 9.1** Two levels of abstraction in a homogeneous multi-agent system where each agent has access to different knowledge bases.

On the lower level, processes are created and destroyed and while they are alive they send messages to each other. Nothing of this kind takes place on the level of logic programs. Entities such as processes, messages, and mailboxes do not exist there – they have been abstracted away. Drawing a line between these two levels is another way to express the distinction between declarative and procedural that so often appears in the story of Prolog. "What?" is specified at the level of logic and the corresponding "How?" is handled on the level of communicating toplevel processes.

The reason this works is that it is very similar to how pure Prolog explores each branch of a search tree independently from other branches. Web Prolog, running pure Prolog on the Prolog Web, explores each *node* independently from other nodes.

From the caller's point of view, a call to `rpc/2-3` in the body of a rule is just another subgoal to process, another search tree branch to traverse, only this time in a Prolog context different from the context of the caller.

This two-level reading is a paradigmatic property. When the sequential partition is pure, the Prolog Web admits both a *logical* reading – in which distributed computation is understood as inference over distributed knowledge bases – and a *procedural* reading – in which the same computation is understood as coordinated message exchange among network processes. Sometimes, when actors and side effects are involved in a fundamental way, only the procedural reading remains. That both readings are available at all in a distributed setting, and that the programmer can move between them, is one of the defining characteristics of the Trinity as a programming paradigm.

### 9.1.2 Unification as the universal data contract

The two-level structure would remain an architectural curiosity if the interface between client and node required the kind of "contract work" that characterises conventional web integration. In a typical web API, one reads documentation to learn an endpoint's URL and parameters, inspects a response schema (often JSON with nested fields), writes deserialisation code that maps responses into application structures, and then maintains that glue as the API evolves. The fragility is structural: three logically coupled things are kept separate – the request shape, the response shape, and the rule that determines whether a response satisfies a request. Conventions such as OpenAPI, JSON Schema, and versioning try to restore this coupling, but only indirectly and after the fact.

In Prolog, unification makes the coupling direct. A goal sent to a node is simultaneously a request for information, a specification of the expected structure of answers, and the compatibility test between the two. The query

```
?- employee(Name, Dept, salary(Min, Max)).
```

both asks for employees and states that answers must have exactly this structure. If the node contains `employee(alice, engineering, salary(80000, 120000))`, the goal unifies and bindings flow back. If the node only defines `employee/2`, the query fails immediately: the mismatch is not a downstream runtime error buried in deserialisation code, it is caught at the point of invocation. The query *is* the contract.

Projection is trivial: a client can ask only for what it needs. Evolution is structural: if a node moves from `employee/2` to `employee/3`, old clients do not silently receive nulls; their queries simply fail.

Composition is native: constraints combine by conjunction, for example

```
?-employee(Name, engineering, salary(_, Max)), Max > 100000
```

The combined query can be executed entirely on the node or split between remote and local evaluation. And because a Prolog predicate is inherently relational, a

single relation supports many access patterns depending on which arguments are instantiated. The client chooses the mode at query time; the node simply unifies and enumerates. A conventional HTTP API needs separate endpoints for each of these patterns; in Prolog they are all the same contract, expressed once, as a goal.

### 9.1.3 Backtracking as pagination

The structural correspondence between Prolog backtracking and HTTP pagination is not a surface-level adaptation but a genuine affinity. Both instantiate the same abstract idea: incremental enumeration under external control. A paginated endpoint exposes a procedure for generating results and a protocol for consuming them in slices; a Prolog toplevel actor does the same, with choice points serving as the resume mechanism and a "next" message as the page-turn signal.

Chapter 4 developed the operational details: the spectrum from naive re-execution through cached-process resumption to explicit pid-based continuation, and the role of the logical update view in providing snapshot-consistent enumeration. What matters here is the paradigmatic observation. Unification provides the data contract, backtracking provides the enumeration model, and HTTP or WebSockets provides the transport – with each layer reinforcing the others rather than working against them. This three-way alignment is one of the reasons we regard the Trinity not merely as a practical engineering effort, but as a paradigmatic fit between logic programming and the architecture of the Web.

## 9.2 From compatibility to interaction portability

The Prolog community has long understood portability as a matter of *compatibility*: the extent to which different systems happen to accept the same programs and produce similar results. Efforts such as the ISO Prolog standard have addressed this at the level of the core language. Yet, as surveyed in *Fifty Years of Prolog and Beyond*, portability issues remain widespread beyond the core – in libraries, constraint systems, concurrency models, and interfaces to the outside world.

Compatibility, even when it covers both syntax and behaviour, is an *empirical* property: it depends on the contingent alignment of implementations, and it is easily broken by innovation. It asks whether two systems happen to agree, not whether they are obliged to. The Prolog Web sharpens the problem further: in a network of nodes implemented on top of different Prolog systems, the question is no longer just whether two systems accept the same program and produce the same answers, but whether they preserve the same observable *interactions* – the same patterns of queries, answers, and message exchanges as seen by clients and peer agents.

In this work, we adopt a different perspective. Rather than treating portability as empirical compatibility between systems, we treat it as a designed property of programs, defined over their observable interactions.

> **Interaction portability.** A program is portable across a class of execution environments if its observable interaction behaviour – the patterns of queries, answers, and message exchanges it can produce – is invariant under relocation between those environments.

This shift has several important consequences. First, portability becomes relative to a well-defined class of environments: a design decision, not an accident of history. Second, interaction rather than textual acceptance becomes central: two systems may accept the same program and yet differ in observable interaction behaviour, and under this notion of portability such differences determine whether portability holds. Third, portability must be *designed* rather than inferred: it requires explicit control over the aspects of execution that affect observable interactions.[1]

The Trinity instantiates this stronger notion directly. Web Prolog enforces inexpressibility: the language is restricted so that client-supplied code cannot depend on uncontrolled or implementation-specific features. Agents and nodes structure execution around message passing and explicit interaction, making behaviour visible and comparable across environments. And node profiles define classes of environments: they provide the reference points relative to which interaction portability is assessed.

### 9.2.1 Node profiles as interaction contracts

If interaction portability is meaningful only relative to a well-defined class of execution environments, then those classes must be made explicit. Chapter 3 introduced node profiles as capability declarations – specifications of what language features and web APIs a node provides. Here we can see them as something stronger: each profile defines not just a set of capabilities, but an *interaction contract* that determines what observable behaviour must be preserved when a program moves between nodes that implement the same profile.

What varies across profiles is not merely what a node can do, but what counts as equivalent behaviour. At each level, the class of observable interaction that must be preserved grows wider, and the agreement required between implementations grows tighter:

---

[1] The distinction drawn here has a parallel in linguistics, where *semantics* concerns the meaning of expressions in isolation, while *pragmatics* concerns how meaning is realised in interaction – including phenomena such as turn-taking, feedback, and conversational structure. What we call interaction portability is closer to a pragmatic than a semantic criterion: it requires not merely that two environments assign the same meaning to a program, but that they produce the same patterns of exchange when that program is used. Traditional compatibility, by contrast, is already partly semantic; the real gap it leaves is at the interactional level.

- **RELATION.** Observable interaction consists of the set of answer substitutions returned for a given query. Two RELATION nodes are interchangeable if they produce the same answers for the same goals. No assumptions about execution order, session state, or continuation need be preserved.
- **ISOBASE.** Observable interaction extends to the *ordering* and *incremental delivery* of answers. Two ISOBASE nodes must agree not only on which answers exist, but on the order in which they are produced and the ability to suspend and resume enumeration. Portability at this level requires that choice point semantics and continuation handling are compatible across implementations.
- **ISOTOPE.** Observable interaction extends to side effects within a session: I/O, dynamic database updates, and the execution of client-supplied code. Two ISOTOPE nodes must agree on the available built-in predicates and on the behaviour of stateful operations – not merely on what answers are produced, but on what effects are produced along the way. Portability at this level is the most demanding within the sequential partition.
- **ACTOR.** Observable interaction is defined in terms of message exchanges, mailbox semantics, and process lifecycle. Two ACTOR nodes must preserve the same ordering guarantees on message delivery, the same failure-propagation semantics, and the same lifetime discipline for spawned processes. Portability here requires agreement not about answers to queries, but about the dynamics of concurrent interaction.

These profiles form a hierarchy of increasingly demanding interaction contracts. What is sufficient for RELATION – agreement on answer sets – is not sufficient for ACTOR, which requires agreement on the dynamics of interaction itself.

Under this view, portability is no longer a fragile byproduct of partial standardisation. It becomes a first-class design principle: programs are written against a profile rather than a specific implementation, their meaning is defined in terms of observable interactions, and systems are free to innovate internally as long as they preserve the interaction behaviour required by the profile. As shown in Section 4.3.3, when a program is self-contained and the target environment implements the required profile, this principle has a concrete operational payoff: relocation becomes immediate, with no adaptation step required.

## 9.3 Agents as the primary abstraction

At the centre of this design space stands the agent. In the Trinity, the agent is not an optional library construct layered atop another abstraction. It is the primary unit of structure.

Actors are agents. Toplevels are agents. Nodes are agents. Statechart processes are agents in structured disguise. Even external systems – whether web browsers, services, or AI components – are treated as agents when interfacing with the Prolog Web. A node is not merely a web server that happens to execute Prolog goals; it is a situated participant that exposes capabilities under a policy, hosts durable knowledge

and services, and mediates between clients, resident actors, and the surrounding network.

This agent-centric stance contrasts with other dominant structuring principles. In object-oriented systems, the object is primary. In functional architectures, the function dominates. In microservice ecosystems, deployment units often eclipse semantic structure. The Trinity instead adopts a process-and-message discipline in which identity, mailbox semantics, and lifecycle policies define the core. Logical reasoning occurs *within* agents; distribution occurs *between* agents; behavioural templates constrain *classes* of agents; and statechart hierarchies govern their reactive control.

The choice has far-reaching implications. It aligns naturally with distribution-first semantics: because the agent is already a process with a mailbox, the step from local to distributed execution does not require a change of abstraction. It supports fault containment and supervision: agents fail individually, and their failure is observable through links and monitors. It allows sequential reasoning and reactive control to coexist: the same agent may answer queries in its sequential capacity and participate in actor protocols in its concurrent capacity.

The agent-centric stance also supports a literal multi-agent reading. Once nodes host durable processes that interact over published protocols, the Prolog Web becomes a web-native multi-agent system in the architectural sense: a population of autonomous processes coordinating by message exchange. This is not a claim about rationality or intention in the AI sense. It is a concrete decomposition principle: each agent has a mailbox, a lifecycle, and a locally scoped program and database; global behaviour emerges from explicit communication under failure and timing constraints.

As Joe Armstrong described the Erlang approach (Armstrong, 2007):

> Our applications are structured using large numbers of communicating parallel processes. We take this approach because [...] it provides an architectural infrastructure – we can organize our system as a set of communicating processes. By enumerating all the processes in our system, and defining the message passing channels between the processes we can conveniently partition the system into a number of well-defined sub-components which can be independently implemented, and tested.

Armstrong's description applies directly to the Prolog Web. But in the Trinity, the process-and-message discipline is not the whole story – it is held in place by a tighter structural property.

### 9.3.1 Mutual constraint

The three components of the Trinity are mutually constraining – a property more characteristic of a paradigm than a toolkit. The language profile constrains the agent model: because clients execute a restricted fragment, agents need not defend against arbitrary meta-calls, and the security posture can be structural rather than procedural. The agent model constrains the network substrate: because agents have

explicit identities, mailboxes, and lifecycles, the interaction protocol must support naming, session tracking, and failure notification – requirements that are reflected in the design of the HTTP and WebSocket layers. And the network substrate constrains the language profile: because programs must cross transport boundaries, the language is shaped so that goals, answers, and code fragments are serialisable as Prolog terms without loss of meaning.

No single component can be redesigned independently without affecting the others. This tight coupling is not a fragility; it is the source of the design's coherence. It is also why the Trinity is not well described as "Prolog plus actors plus HTTP". The components do not merely coexist; they shape each other.

### 9.3.2 Capability-oriented security

One consequence of this mutual shaping is the security model. Because the language profile, the agent model, and the network substrate constrain each other, safety does not have to be imposed from outside — it arises from the architecture itself.

The structural mechanisms that make the Prolog Web safe to open — inexpressibility, invisibility, and containment — are developed in Section 4.5. What matters paradigmatically is that these mechanisms constitute a *capability-oriented* security model: authority is conveyed by the possession of references, not granted by external access-control lists. If a program does not hold a reference to an actor, service, or node, it cannot reach that entity. The principle can be stated compactly: *possession of a reference is the authority to use it*.

What is significant is not that the Prolog Web has a security model, but that its security model is *structural*. Inexpressibility is enforced at the language boundary: dangerous operations cannot be named in the client fragment. Invisibility is enforced by the address space: a pid that has never been revealed cannot be guessed. Containment is enforced by lifetime dependency: when a session ends, its actor subtree disappears. Safety arises from the architecture, not from runtime checks scattered through the code.

This structural character aligns the Prolog Web with a family of contemporary execution environments — browser-based JavaScript, WebAssembly, WASI — that share the same posture: code executes inside a deliberately bounded environment and acquires authority only through explicitly provided references. Traditional Prolog systems are designed around maximal expressiveness; Web Prolog inverts the default, treating expressive power as owner-controlled capability that can be selectively granted rather than silently assumed.

### 9.3.3 Five complementary views

The agent-centric design refracts into several complementary perspectives, each emphasising different linked entities and activities:

**The Prolog Web of source code.** URIs link programs to programs. Nodes expose source code over HTTP; users and tools can "view source" by following links.

**The Prolog Web of query solutions.** URIs link callers to answer streams. A client asks a node to solve a goal and incrementally enumerates solutions. Interaction is sequential and pull-based.

**The Prolog Web of actors.** Pids link live processes with mailboxes. Interaction is asynchronous: `!/2` sends, `receive/1-2` receives. Linking and monitoring make lifetimes explicit.

**The Prolog Web of behaviours.** Actors implement behaviours – protocols specifying message syntax, semantics, and timing. The behaviour is the contract; the actor is the running entity.

**The Prolog Web of nodes.** Nodes are the "places" of the system: they store and serve code, host actors, enforce policy, and advertise capabilities. They are points of execution and accountability.

These are not competing architectures but complementary perspectives on the same system. A Prolog Web application typically involves all five simultaneously: source code that is browsable, goals that are queryable, actors that are addressable, behaviours that are composable, and nodes that are accountable.

## 9.4 The logical foundation: purity, belief, and scoped inference

The agent-centric view supplies the structural principle; the logical foundation supplies the semantic one. The Trinity is not merely a concurrent system that happens to use Prolog syntax. Logic programming plays a constitutive role, and the key to understanding that role is the notion of a *pure Prolog Web*.

### 9.4.1 Pure Web Prolog and the pure Prolog Web

As noted in Chapter 4, `rpc/2-3` retains the logical purity of the predicates it calls. This allows us to form the notion of *pure Web Prolog*:

$$\textit{Pure Web Prolog} = \textit{Pure Prolog} + \textit{rpc/2-3}$$

Purity here means purity of answer substitutions as logical consequences; timeouts and connection failures may truncate the answer stream but do not corrupt it. The union of all partitions of the Prolog Web written in pure Web Prolog forms what we

think of as the *pure Prolog Web*. With a large number of nodes running pure Web Prolog programs interlinked to other pure Web Prolog programs, we may *in theory* create a Web of Logic on top of the conventional Web – a layer that can *in principle* grow as large as we want it to grow while still adhering to the formal semantics of the pure subset of the Prolog language. In practice, this may never happen, but the *architecture* is there.


**Fig. 9.2** A pure Prolog Web spanning the globe remains aspirational, but the architecture does not preclude it.

### 9.4.2 Epistemic reading and scoped inference

If agents are the primary abstraction and the pure Prolog Web gives them a logical character, then a call such as `rpc(URI,Q)` admits a natural epistemic reading: it asks whether the agent at `URI` *believes* that `Q`. A clause such as

```
human(X) :- rpc('http://n1.org', human(X)).
```

can, in the syntax of predicate logic enhanced with a higher-order predicate *believes/2*, be formulated as

$$\forall x[believes(n_1, human(x)) \rightarrow human(x)]$$

It makes perfect sense to say that one agent believes that `human(socrates)` is true while another does not:

```
?- rpc('http://n2.org', human(socrates)).
true.
```

```
?- rpc('http://n1.org', human(socrates)).
false.
```

And the following query asks whether two agents *agree* that `human(X)`:

```
?- rpc('http://n1.org', human(X)),
   rpc('http://n2.org', human(X)).
X = plato ;
X = aristotle.
```

The epistemic reading is not the only one available. More conservatively, the URI can be understood as specifying the *scope* of inference – the knowledge base with respect to which a query is to be evaluated. As argued in (Kifer et al., 2005), scoped inference is mandatory for realising Negation As Failure on the Web, because applying any form of the closed world assumption requires knowing the entire knowledge base first – something that is not possible for the Web since its boundaries cannot be clearly delineated. A third and more practical view is simply that `rpc/2-3` invokes a goal in the context of a remote *module*, with data encapsulation that prevents the mixing of predicates with the same name and arity. For the pure Prolog Web, the three views – epistemic operator, scoped inference, remote module – converge.

What about `rpc/3`? Code injection via `load_*` options does not jeopardise purity provided the final composition of the program executed is itself pure. The `limit` option is a *pragma*: it specifies the granularity of the conversation between client and node but has no effect on the meaning of the query.

### 9.4.3 The less pure Prolog Web

Not all of the Prolog Web is pure. ISOBASE nodes may make use of control constructs such as cut, and ISOTOPE nodes go further with side-effecting predicates, I/O, and dynamic database updates. What statelessness constrains is persistent state modification within a single HTTP request – a protocol-level restriction, not a language-level one.

More fundamentally, the concurrent Prolog Web – the world of the ACTOR profile – is inherently procedural. Actor-based programming involves process creation and destruction, message ordering, selective receive, and failure propagation – none of which have declarative readings. A realistic application will typically mix profiles freely, using pure queries for declarative reasoning and actors for coordination and state.

This spectrum from pure to impure is itself a paradigmatic feature. Rather than forcing a choice between declarative and procedural – a tension that has shadowed Prolog since its inception – the Trinity makes the boundary explicit and navigable. The programmer can reason about which parts of a system admit logical readings and which require procedural ones, and can move between the two with architectural awareness rather than accidental drift.

## 9.5 The distribution-first model

Web Prolog adopts a distribution-first view: concurrency and distribution are not anomalies to be managed but the fundamental reality of computation. Local execution is simply the case where network latency happens to be zero. The Actor Model views computation as inherently distributed, with independent actors communicating asynchronously; sequential processing is a special case. For Web Prolog, treating distribution as normal rather than exceptional simplifies the programming model and aligns naturally with web-scale deployment.

The formal foundation for this stance comes from Svensson et al. (2010), who proposed a unified semantics for Erlang-like systems. Their key insight is to eliminate the distinction between local and remote communication: all messages travel through a single conceptual mechanism, and side-effecting operations are handled uniformly by a node controller abstraction. This makes placement a configuration detail rather than a semantic distinction. Web Prolog can adopt this cleaner semantic story from the outset, rather than being constrained by decades of backward compatibility as Erlang is. Svensson, Fredlund, and Benac Earle themselves hinted at this when they noted that "perhaps the language for this semantics is not Erlang, maybe there is another language waiting around the corner." We believe the language waiting around the corner is Web Prolog.

A useful way to read the distribution-first stance is through the classic goal of *distribution transparency*: making the distribution of processes and resources invisible to applications. The Prolog Web does not aim for transparency by hiding the network behind a magic API; rather, it aims for a disciplined form in which the *same* conceptual operations — `spawn`, send, monitoring, `rpc/2-3` — apply locally and remotely. Latency and communication-failure probabilities change with distance, but the programming model does not bifurcate into a "local" and a "remote" language. What makes this disciplined rather than naive is the profile hierarchy: the profiles specify exactly which guarantees hold across the local/remote boundary, so that transparency is a known contract rather than a leaky abstraction.

The combination of Prolog's execution model with this actor semantics can be given a clean, modular formal account. Conceptually, we extend the unified actor semantics with a disjoint state component for each process: in addition to its mailbox and process control state, an actor carries a *Prolog engine state* consisting of a goal stack and a current substitution. The Erlang-style rules manipulate only processes and mailboxes; the Prolog rules touch only goals and substitutions. Because these state components are disjoint, the two rule sets can be interleaved without conflict. Backtracking remains local to a single actor's engine; interaction between actors is always mediated by messages. A full development of this layered semantics is beyond the scope of this chapter, but the composability of the two rule sets is what justifies treating the sequential and concurrent partitions as genuinely orthogonal.

### 9.5.1 The browser as a Web Prolog runtime

If placement is a configuration detail, then the distribution-first model extends naturally to the client side. A browser equipped with a Web Prolog runtime is not a node – it does not accept incoming connections – but it is a first-class participant in the Prolog Web, able to use the same interaction primitives available to server-side code. The programmer should be able to use synchronous calls when blocking is acceptable, non-blocking request–response calls when responsiveness matters, and actor-style interaction for cases that genuinely require long-lived conversational state – all without being exposed to transport-level details.

The longer-term ambition suggested by this design is Prolog on both sides of the Web. At the same time, this should not be confused with an attempt to hide the Web or to force all clients into a single language. Many developers will continue to prefer writing front-ends in JavaScript and communicating with Prolog back-ends through explicit web APIs. The Prolog Web should make such APIs visible, well-specified, and robust. A Prolog-everywhere experience may make them largely invisible in the common case, but engineering robustness requires that they remain accessible.

## 9.6 Control as a first-class dimension

Consider the famous equation coined by Robert Kowalski:

Algorithm = Logic + Control

The kind of control we have in mind here is different from search control. Conversational and reactive systems are hard not primarily because they require sophisticated algorithms, but because they require fine-grained *interaction control* – the kind that governs turn-taking, orchestration, and long-lived dialogue flow. In the Trinity, this motivates the combination of three programming models: logic programming for knowledge and inference; actor-based programming for concurrency, distribution, and interaction; and statecharts for explicit, inspectable control structure. The three are orthogonal: each addresses a dimension of system design that the others leave open.

### 9.6.1 Governance of interaction

If the Trinity were only a story about distributed queries and actor concurrency, it would leave an important practical question unanswered: how does one *govern* complex interaction so that it remains inspectable, auditable, and robust? Protocol sequencing, timeouts, retries, cancellation, interruption, recovery, and the disciplined handling of exceptional cases – these are the concerns that dominate real web appli-

cations and agent systems, and none of them are addressed by logic or concurrency alone.

Statechart actors are the Trinity's response. The idea is not to replace actor programs with statecharts, but to offer a structured substrate for reactive behaviour in which states, transitions, events, timers, guards, and recovery paths are explicit and therefore reviewable. Statechart actors turn the actor model from a concurrency substrate into a *governable interaction platform*.

On the level of notation, this can be realised in more than one way: a Prolog-friendly representation for SCXML-style charts, with translation to and from SCXML when interchange is useful; or SCXML itself as the control notation with Web Prolog as the language of guards, actions, and data. Either way, the relationship follows a familiar web pattern: a declarative control layer whose behaviour is scripted in a general-purpose language.

### 9.6.2 A platform for the Agentic Web

Chapter 8 placed the Trinity in a broader movement: software is becoming increasingly agentic. The Trinity's claim is that the Prolog Web is a natural substrate for such systems because it integrates three capabilities that agentic applications repeatedly need: a web-native network of nodes, a concurrency model for durable interactions, and a symbolic language for explicit representation and constraint.

The Agentic Web framing supplies the *application pressure*: conversational and tool-using agents need explicit protocol state, recovery from partial failure, and coordination under bounded waiting. Statecharts supply the *governance mechanism*: inspectable control with timeouts, retries, and fallback paths as first-class structure. Logic programming supplies the *symbolic substrate*: knowledge, constraints, and goal-directed inference remain available inside agents and can be shipped, composed, and queried across node boundaries.

This synthesis motivates a pragmatic neuro-symbolic stance. Statistical language models are useful components in agentic systems, but they are not reliable as autonomous governors of protocol and state. The Trinity treats statistical components, when present, as powerful but fallible modules, and places responsibility for durable behaviour in explicit symbolic structures. "Agentic" does not merely mean "chatty". It means *durably interactive under explicit rules*, in an open network where failure and delay are normal.

## 9.7 Executable logic on the Web

A recurring posture in web architecture is to treat logic as primarily *representational*: a way to publish structured meaning, express constraints, and support inference over shared data. The Semantic Web strand exemplifies this orientation. It emphasises

common identifiers, interoperable data models, and well-defined entailment regimes. These are substantial achievements, but they leave open a practical question: where does executable reasoning live, and how is it composed across independently hosted sites?

The Trinity's distinctive move is to treat logic on the Web as *executable* in a disciplined sense. A node is not only a publisher of facts; it is a host for relational services, rules, and interaction protocols that can be invoked, combined, and enumerated as computations. In the sequential partition, a goal can be shipped to a node and answered nondeterministically, with unification as the data contract and backtracking as the enumeration model. In the concurrent partition, long-lived agents coordinate by message exchange while embedding explicit symbolic structure in the messages they exchange and the constraints they apply.

Once this substrate exists, "logic on the Web" is no longer confined to representation and offline reasoning. It becomes a first-class component of live systems: callable, composable, inspectable, and governable. This executable substrate is what makes it natural to treat the Trinity as a form of symbolic middleware for the Web.

### 9.7.1 Positioning relative to the Semantic Web

The Prolog Web and the Semantic Web are similar in spirit: both treat the Web as a computational environment, both rely on global naming and logic-based machinery, and both aim at decentralisation. But they emphasise different things. The Semantic Web places its centre of gravity on interoperable data models and shared semantic commitments; the Prolog Web places its centre of gravity on executable relational services, process-level structure, and web-native agents.

| **The Prolog Web** | **The Semantic Web** |
|---|---|
| Uses the same language for querying and rule-based computation | Typically separates data representation (RDF/OWL) from querying (SPARQL) and application logic |
| Provides a process and service architecture as part of the proposal | Provides standards for interoperable data and ontology semantics, but does not prescribe a process architecture |
| Supports agent-style computation through long-lived actors and message passing | Has strong support for shared ontologies and open-world reasoning through standardised languages |
| Shared meaning is typically implicit in executable predicates; shared ontologies are optional | Interoperability is centred on shared identifiers, vocabularies, and ontology constraints |
| Well suited as a programming and orchestration layer around heterogeneous services | Well suited as a lingua franca for knowledge interchange |

**Table 9.1** Comparing the Prolog Web and the Semantic Web.

The Semantic Web asks: *how can independently published data be understood and combined?* The Prolog Web asks: *how can independently hosted logic be executed and composed?* The two questions are complementary. RDF, OWL, and SPARQL provide machinery for representing and querying structured knowledge. The Trinity adds an execution substrate: a way to run queries, host rules, ship code, and maintain long-lived interaction under explicit operational rules. The paradigmatic claim is that the Trinity fills a gap the Semantic Web has long acknowledged: the gap between representation and execution, between knowing and doing.

Although this section uses the Semantic Web as a concrete reference point, most of the underlying issues are not specific to RDF, OWL, or W3C standards. Property-graph systems and large-scale knowledge-graph platforms have converged on similar concerns. The Prolog Web addresses this broader design space, offering an executable logic-based substrate that can sit alongside different graph representations rather than competing with any single one. This is consistent with the multi-stack architecture for the Semantic Web proposed in (Kifer et al., 2005; Horrocks et al., 2005).

## 9.8 Comparison to other approaches

The claim that the Trinity constitutes a candidate paradigm invites a natural sceptical response: has this not been attempted before? Three precedents are close enough to warrant explicit comparison.

### 9.8.1 Linda and tuple-space coordination.

Linda (Gelernter, 1985) introduced coordination through a shared associative memory, with pattern matching over tuples as the retrieval mechanism. The Trinity shares the emphasis on pattern matching as coordination, but the Prolog Web is a network of active agents with explicit identities and lifecycles, not a passive shared store. Coordination is directional: a client submits a goal to a named node and receives answers through a well-defined protocol. This directedness makes reasoning about authority, lifetime, and failure tractable.

### 9.8.2 Concurrent Prolog and the committed-choice languages.

Concurrent Prolog (**?**), Parlog (Clark and Gregory, 1986), and GHC (**?**) showed that SLD-resolution and concurrency are not incompatible, but they paid a significant price: the declarative reading is severely weakened by the commitment mechanism, and backtracking is absent from the concurrent partition. The Trinity keeps both.

The sequential partition retains full Prolog semantics; the concurrent partition adopts Erlang-style message passing, making the non-declarative character of concurrency explicit. The two are deliberately orthogonal: backtracking does not roll back mailboxes, and actor communication does not interfere with proof search.

### 9.8.3 Oz/Mozart and the multiparadigm kernel language.

Oz (Smolka, 1995), implemented in the Mozart system (van Roy et al., 2003), is the closest and most instructive precedent. Like the Trinity, it combines logic programming, concurrency, and distribution in a single coherent design. Distributed Oz (van Roy et al., 1997) adds network-transparent mobility and fault tolerance, and the textbook (van Roy and Haridi, 2004) provides an unusually clean formal account through a hierarchy of kernel languages. The comparison therefore cannot rest on missing features; it must rest on design stance.

The decisive difference is that Oz replaces Horn clause syntax with a fully compositional, higher-order kernel language that accommodates all its paradigms uniformly. The result is a coherent and expressive design, but one that is no longer Prolog in any meaningful sense: it abandons unification-driven clause selection, backtracking, and the program-as-database identity that gives Prolog its distinctive character. The Prolog community has not adopted Oz, and the distance is not bridgeable by a compatibility layer. The Trinity, by contrast, is defined as a *profile* of Prolog. It extends ISO Prolog with concurrency primitives and distribution infrastructure while remaining syntactically and semantically recognisable to anyone who already knows the language. This is a deliberately narrower ambition – but it is the right ambition for a proposal that aims to unify an existing fragmented community rather than replace it with a new one.

A second difference concerns the Web. Oz/Mozart achieves genuine network transparency through a custom protocol over TCP, but nodes are not URIs, processes are not addressable by HTTP clients, and services are not discoverable by web tooling. The Trinity takes the opposite stance: HTTP and WebSocket are the primary transport substrate, and the addressability of nodes by URI is a design commitment. The result is a system less general than Oz in the space of distribution models but more naturally embedded in the Web as it actually exists.

## 9.9 Why the Trinity fits Prolog

The Trinity architecture is not merely implemented *in* Prolog; it resonates with properties of the language that make the fit unusually tight.

Unification gives the ecosystem a universal data contract: structured terms serve simultaneously as data structures, messages, and query templates, and the same mechanism that matches a goal against a clause head also matches incoming mes-

sages against expected patterns. Nondeterministic computation provides a disciplined enumeration model that maps naturally onto Web-style incremental consumption. The proximity between code and data means that predicate shipping and remote goal execution can be expressed using ordinary language constructs rather than elaborate serialisation frameworks. And Prolog's strengths in symbolic reasoning and explanation-oriented computation complement the broader ambition of supporting agents capable of reasoning about structured knowledge.

What tends to disappear when the pattern is transplanted to other languages is the degree of *semantic unity*. In Prolog, terms, clauses, goals, and substitutions form a single conceptual framework supporting knowledge representation, communication, and computation simultaneously. In most other languages these concerns are handled by separate mechanisms: data structures, serialisation formats, rule engines, and orchestration frameworks. The resulting systems may be powerful, but they lack the conceptual compression that Prolog provides. The Trinity is therefore best viewed as a web-native expression of the logic programming paradigm – the paradigm in which Prolog has historically found its most direct realisation.

## 9.10 Summary

This chapter has stepped back from the technical machinery developed in earlier chapters to ask what conceptual pattern emerges when the components of the Prolog Trinity are considered as a whole.

The answer is a candidate paradigm for web-native symbolic computation, characterised by several interlocking and mutually constraining commitments. Two orthogonal partitions – sequential and concurrent – provide a stable design axis that separates declarative reasoning from actor-based interaction. Semantic portability and node profiles replace fragile compatibility with a designed, contract-based notion of what it means for programs to retain their meaning across environments. The agent serves as the primary abstraction, unifying actors, toplevels, nodes, and statechart processes under a single process-and-message discipline – and the five complementary views of the Prolog Web show how this abstraction refracts into a rich but coherent set of perspectives. A logical foundation preserves the possibility of pure, declarative reasoning even in a distributed setting, with the epistemic reading of `rpc/2-3` and scoped inference giving formal substance to what it means to query a web of knowledge bases. Control, in the sense of interaction governance, is treated as a first-class design dimension through the integration of statecharts alongside logic and actors. And the mutual constraint among language profile, agent model, and network substrate – each shaping and being shaped by the others – is what gives the design its coherence and prevents it from decomposing into a loose collection of independent ideas.

Whether this constellation of commitments constitutes a paradigm in the strongest sense of the word is ultimately a judgment that the community, not the author, must make. What we can say is that these commitments are not accidental. They emerged

from a deliberate attempt to integrate logic programming, actor-based concurrency, and web architecture into a coherent whole – and that the result, for better or worse, does not reduce to any of its components.

# Part V
# What's in it for you and your community?

# Chapter 10
# What's in it for the Prolog community?

> Coming together is a beginning; keeping together is progress; working together is success.
>
> *Henry Ford*

The earlier chapters developed the Prolog Trinity ecosystem as a technical proposal: Web Prolog as a disciplined profile, Prolog agents as programmable entities, and the Prolog Web as a network of nodes that supports both sequential query interaction and long-lived concurrent conversations. The remaining chapters turn from architecture to value: what, concretely, would this ecosystem make easier or better for the communities that might choose to build with it?

This chapter focuses on the Prolog community itself: implementers, tool builders, teachers, and users who already care about Prolog's future. Chapter 11 widens the lens to neighbouring communities – logic programming, the Semantic Web, Artificial Intelligence, the Erlang ecosystem, and web programmers – and asks where the Trinity might offer a useful point of contact. Our aim is to be explicit about trade-offs: which problems the Trinity is meant to address, what it might enable, and which costs it would impose.

For more than five decades, Prolog has served as a testament to the power of logic in computation. Yet, as briefly indicated already in Chapter 1, it has also faced persistent challenges that have, at times, constrained its broader impact. The Prolog Trinity ecosystem is conceived, in no small part, as a direct response to these challenges, offering pathways to reinvigorate the language, and amplify its collective strength.

We will again turn to *Fifty Years of Prolog and Beyond*, where a description of the current state of Prolog is given. We allow ourselves to use the collection of logotypes in Figure 10.1 as an impressionistic representation of the state of the Prolog ecosystem as it exists today – the thirteen currently maintained Prolog systems and the communities surrounding them.

As a background to our discussion, the detailed SWOT analysis presented by the authors of FYPB proves particularly valuable. Their analysis systematically exam-


**Fig. 10.1** Logotypes for thirteen Prolog systems: B-Prolog, Ciao, ECLiPSe, GNU Prolog, JIProlog, Scryer Prolog, SICStus Prolog, SWI-Prolog, Tau Prolog, Trealla Prolog, tuProlog, XSB and YAP. Except for Trealla Prolog, these are the systems that FYPB lists as "currently maintained." The matching of systems with logotypes is left as an exercise for the reader.

ines the strengths, weaknesses, opportunities, and threats characterizing the Prolog ecosystem, with the aim of reinforcing its strengths, addressing its weaknesses, seizing available opportunities, and mitigating potential threats.

Among the *strengths* that Prolog has, the authors of FYPB not only remind us about the core features of Prolog such as unification and backtracking, but also highlight the many powerful features, such as tabling, concurrency and constraints, that have been developed over the years. They also remind us about how efficient Prolog systems have become. However, for readers familiar with the systems in Figure 10.1 it should be evident that those strengths are rather unevenly distributed. Some systems offer excellent constraint programming capabilities, others excel at tabling, while others yet focus on concurrency. Some are really fast, others rather slow. XSB's sophisticated tabling mechanisms offer unparalleled performance for certain classes of deductive database queries and logical inferences. ECLiPSe's powerful constraint solvers are indispensable for complex optimization and scheduling problems. SWI-Prolog provides a remarkably comprehensive environment with extensive libraries, making it a popular choice for a wide range of applications. In most comparative speed benchmarks, YAP and SICStus Prolog decisively outperform the other systems.

Occasionally, a new Prolog system emerges that introduces a feature not available in other systems, one that may be difficult or impractical for existing systems to implement. For example, Scryer Prolog and Trealla Prolog have introduced a new form of compact and efficient string representation that emulates a list representation and from the programmer point of view is very much indistinguishable from a list.

As for *threats* and *weaknesses*, the authors of FYPB appear to see *fragmentation* as Prolog's most serious threat. We believe they are right, and the patchwork of logotypes in Figure 10.1 might be seen as a sign of this threat. Add a user base too small to the problems that Prolog faces – the language simply is not popular.

## 10.1 A middle-way approach to systems fragmentation

According to the authors of FYPB, the fragmentation of the Prolog *systems* landscape is indeed a concern since while most of these systems adhere to the ISO Prolog core standard, many differences appear in the added features that are not handled by ISO.

While the diversity is a testament to Prolog's adaptability and the ingenuity within its community, it has historically presented a barrier to collaboration and code reuse. Applications developed in one Prolog system are often difficult to port to another. The traditional pursuit of complete "portability" – the dream of writing Prolog code that runs flawlessly on any system – while laudable, has proven elusive. Fragmentation also leads to difficulties writing textbooks and tutorials, might be a threat to further language standardization efforts, and hinders the development of a healthy ecosystem.

This section explores three distinct strategies for addressing this long-standing challenge. The first is a bottom-up, pragmatic approach that focuses on incremental improvements and standardization of specific features to gradually enhance portability across systems. In stark contrast, the second approach is a top-down, visionary effort to create a single, unified Prolog that would integrate the best features of all existing systems. Finally, we will examine a third, middle-way approach that champions interoperability over uniformity, proposing a federated model where different Prolog systems can communicate and collaborate in real-time without losing their individual identities. This third approach, which is the focus of this book, is our own proposal.

### 10.1.1 Reducing systems fragmentation one bite at a time

Since the publication of FYPB, a group of Prolog implementers have begun to address several of the challenges and risks highlighted in the paper. This effort, undertaken over the past few years, involves steps to reduce systems fragmentation. While it is still an ongoing process, these initial actions appear to represent a growing recognition within the Prolog community of the need to mitigate identified weaknesses and ensure a more cohesive and sustainable future for the language.

*The Prolog Implementers Forum* (with representatives from at least Ciao, ECLiPSe, Logtalk, SWI-Prolog and XSB) is aimed at building consensus on the future of Prolog. In an attempt to address the problem of fragmentation of systems and the code portability problem, forum members have started to try to eliminate some of the differences between systems. Inspired by similar community-driven improvement processes in other programming ecosystems, but tailored to meet the specific needs and challenges of the Prolog community, this has taken the form of *Prolog Improvement Proposals* (PIPs). PIPs are meant to serve as a mechanism for proposing and discussing important enhancements to the Prolog language and its ecosystem. Although not part of the official ISO Prolog standardization effort, they are intended to provide a structured means for documenting, exploring, and eval-

uating new features, libraries, and extensions that may eventually influence future standardization.

We do like the idea of using PIPs as a means of improving the portability of as large a fragment of Prolog as possible, thereby helping to reduce fragmentation across systems. This is a *realistic* approach. However, while undoubtedly worthwhile, the PIP effort addresses only one of the questions raised above, namely: How should efforts to increase portability be organized? The chosen answer is: bottom-up, incremental, and in small steps.

PIPs provide a structured arena for implementers to negotiate changes, preventing systems from diverging in isolation. By aligning on specific areas like library interfaces, they shrink the "portability gap" and reduce the practical costs of fragmentation. Acting as a bottom-up proving ground for features before potential ISO standardization, PIPs ensure formal standards connect to real-world usage. They do not promise a sweeping vision, but rather a mechanism for practical convergence that keeps diverse systems compatible.

However, this approach faces significant hurdles. Progress is notably slow; despite seven drafted PIPs, no decisions have been reached, and community engagement has stalled. More critically, the effort fails to address the broader question of Prolog's long-term evolution. While the proposals are pragmatic, they lack a unifying vision, and this very absence of direction likely contributes to the sluggish pace. Without a shared roadmap for the future, incremental efforts risk becoming disconnected, piecemeal fixes rather than steps toward a larger goal. Vision acts as a necessary coordinator and motivator, inspiring contribution and helping to prioritize which improvements matter most. When this compelling narrative is missing, discussions fragment, contributors drift away, and even well-designed initiatives lose the momentum required to succeed.

### 10.1.2 One Prolog to rule them all

For an especially visionary perspective, we may turn to one of the contributions in *Prolog: The Next Fifty Years*, where Gupta et al. answer the question – asked in FYPB but never answered – of whether it makes sense to aim for a unified language with a resounding "yes." They propose an ambitious vision of a system that seamlessly integrates tabling, constraints, concurrency, and parallelism within a single coherent framework.

The idea of a unified system is not new. In a 2016 discussion on Stack Overflow,[1] Jan Wielemaker proposed:

> Ideally we should sit together and assemble a new system that combines the speed of YAP with the tabling of XSB, the constraints of ECLiPSe, and the environment and interfaces of SWI-Prolog.

[1] `https://stackoverflow.com/questions/35668475/prolog-vs-erlang-and-other-functional-languages`

Let us give this hypothetical system a name. Interestingly, by permuting the initial letters of the component systems' names, one arrives at a playful and memorable moniker for the new platform: SEXY Prolog!

The approach of Gupta et al goes even further than Wielemaker though, as they also aim to incorporate *Answer Set Programming* (ASP) into Prolog, or more specifically s(CASP), which combines the strengths of ASP and constraint logic programming in a top-down, query-driven model of computation that includes support for explanation generation. The hope is that this leads to a sophisticated evolution that consolidates various extensions and paradigms into a cohesive programming environment even capable of commonsense reasoning. This integration is claimed to be designed not only to expand Prolog's expressive power but also to align it more closely with the evolving demands of artificial intelligence.

So what happened to SEXY Prolog? Its spirit of integration lives on, and SWI-Prolog has been repeatedly described as a system that "glues together" many Prolog research and application innovations into a broadly usable environment. In recent years, SWI-Prolog has incorporated advanced features such as CLP (inspired by ECLiPSe – the prototypical CLP-oriented Prolog system), tabling (a key feature of XSB – but in an implementation that according to the manual is a "first prototype" requiring further enhancement to rival a mature system such as XSB."), and for speed, Just-In-Time indexing (for which the manual says that "YAP's indexing has been the inspiration for enhancing SWI-Prolog's indexing capabilities."). On top of this, SWI-Prolog even has an implementation of s(CASP).[2]

In this way, users do not need to switch between multiple systems for different features; they can stay in SWI-Prolog and get "good enough" versions of everything. However, it seems that for Gupta et al, "good enough" is, in fact, not good enough. What matters is not merely the presence of advanced features, but their *seamless* integration – a standard that, in their view, SWI-Prolog probably does not yet meet. They therefore contend that research "to realize a system where all these advanced features are efficiently and seamlessly integrated and available in a single system must continue."

The approach of Gupta et al is extremely interesting, but we suspect that it is not likely to result in anything practically useful any time soon. It would likely take decades for a unified language of this kind to materialize and become useful. The question is also whether the result, if it materializes, will still be Prolog? We believe it is more likely to end up as another Mercury, Oz or Picat. They are excellent programming languages, but they are not Prolog.

### 10.1.3 The middle way: interoperability as a federating force

The history of programming language standardization is largely a history of unification attempts: efforts to define a single language, a single type system, a single

[2] https://github.com/SWI-Prolog/sCASP

runtime, or a single ontology that all participants must adopt. Such efforts have produced valuable results – ISO Prolog, the W3C Semantic Web stack, and the JVM are notable examples – but they also carry a characteristic risk. Unification demands consensus on internal design decisions; the broader and deeper the consensus required, the harder it is to achieve and maintain. In the Prolog world specifically, the fragmentation of implementations and the difficulty of converging on shared standards have been persistent challenges.

The Trinity takes a different approach. It defines shared *interaction surfaces* – profiles, protocols, and APIs – while leaving internal implementation choices free. A node need not run a particular Prolog system; it need only conform to a profile that specifies which predicates, transports, and behavioural contracts it supports. Different nodes may implement the same profile in different ways, with different optimisations, different libraries, and different internal architectures. What is standardised is the *interface*, not the *engine*.

This is federation in a precise sense: independent parties agree on how to communicate without agreeing on how to think. The agreement is operational (we will exchange messages in this format, over these transports, with these semantics) rather than ontological (we will represent the world in the same way). This is also why the Trinity can position itself as scaffolding around existing Prolog systems rather than as a replacement for any of them.

Figure 10.2 illustrates the concept in concrete terms. Rather than collapsing the distinctive strengths of SWI-Prolog, ECLiPSe, XSB, and YAP into a single uniform system, we envision an ecosystem where each system is wrapped in a node implementation that allows it to converse with others. The diversity of approaches is preserved, yet they can all contribute to a common fabric of cooperation. While participation is voluntary, a node interface of this kind is indispensable for any system aspiring to take part in this shared Prolog Web.

As demonstrated in the proof-of-concept implementations accompanying this book, Web Prolog nodes owned by different entities and running in different locations can offload computations to one another and engage in mutual communication. This represents a significant step towards real-time interoperability. However, the discussion so far has been limited to homogeneous cases where nodes run on the same underlying Prolog system. The next challenge is to extend this capability to heterogeneous settings, where nodes may rely on different Prolog systems yet must still achieve the same level of seamless, real-time interoperability.

### Portability vs. real-time interoperability

Much like the work pursued by the Prolog Implementers Forum (PIF), the advancement of this middle way requires cooperation among system implementers. However, while the PIF emphasises *portability* – the ability to move code between platforms offline – our proposal extends this focus to *real-time interoperability*. Portability enables code to move between systems; interoperability enables the systems themselves to interact during execution.

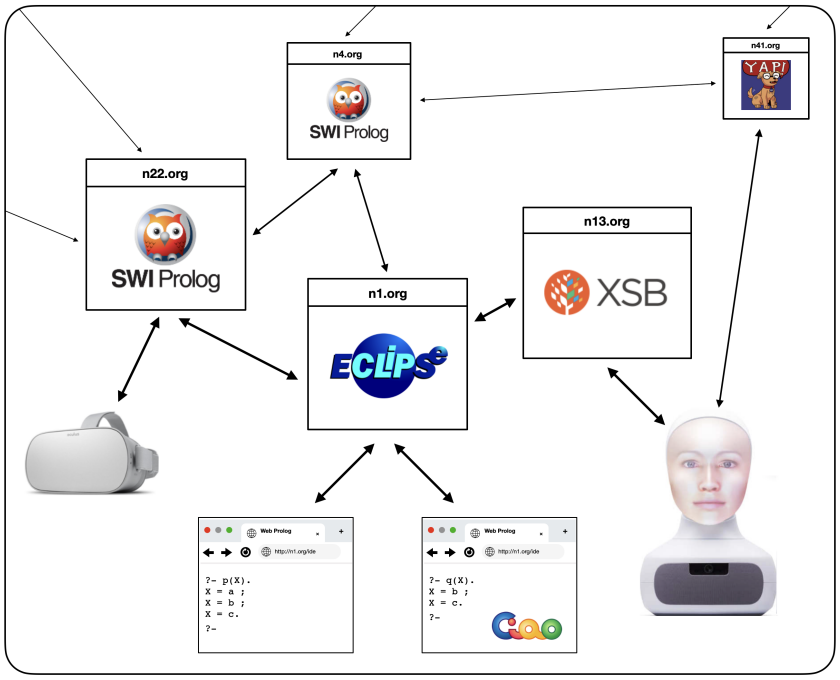



**Fig. 10.2** A federated middle way: interoperability without forcing a single unified system.

Because runtime conformance issues cannot be easily fixed on the fly, the demands placed on Web Prolog systems must be strictly defined. The Prolog Trinity ecosystem offers a vision of *cross-system interoperability as a federating force*. Instead of striving to make all Prolog systems behave identically, Web Prolog functions as a *lingua franca* that enables distinct systems to communicate, collaborate, and interoperate seamlessly.

### Reframing fragmentation as diversification

We emphasise that this approach allows Prolog systems to *retain their differences* on many levels. We encourage a "let a thousand flowers bloom" philosophy: implementers are free to add extensions and optimisations, provided they do not interfere with the Web Prolog layer. Through calls to the host language, a Prolog Web node can still offer the functionality of these unique extensions; indeed, their presence might be exactly why a developer chooses that specific node implementation.

Imagine a future where an application orchestrated by Web Prolog can dispatch a deductive query to a remote XSB engine, feed the results into an ECLiPSe solver, and manage the user interface via SWI-Prolog. Each system contributes its peak

capability, forming a federated ecosystem where developers do not have to be content with "good enough" versions of everything, but can aim for "the best there is."

From this vantage point, the current landscape of the Prolog community changes. What is often labelled as fragmentation can be reinterpreted in a more constructive light: as *diversification*. This shift in framing underscores the presence of a dynamic, albeit loosely coupled, constellation of systems that collectively sustain innovation and resilience.

#### Engineering a collaborative future

Much like the grand vision for a Unified Prolog, our approach entails substantial work. However, it demands less in terms of foundational research and more in terms of disciplined *engineering*. We already possess mature building blocks that need to be consolidated rather than re-invented. Furthermore, *inclusivity* is a core principle. To ensure that the barrier to entry is not insurmountably high – particularly for systems lacking multi-threading – the profile hierarchy proposed in Chapter 3 allows even the least powerful Prolog systems to integrate by conforming to simpler profiles.

Ultimately, the viability of this middle way depends on a symbiotic relationship with the Prolog Implementers Forum. The PIF's efforts to catalogue divergences and clarify semantics form the substrate upon which our interoperability architecture is built. In return, the Trinity ecosystem acts as a unifying testbed: a practical arena where abstract agreements take on technical reality. This offers a gradual path toward unification, where interoperability precedes standardisation, allowing the Forum's work to crystallise into a shared, operationally validated Prolog core that systems can adopt without sacrificing their individuality.

If the Prolog Web succeeds, it will not be because every Prolog system has been replaced by a single implementation, but because many implementations have adopted compatible interaction surfaces. Success is measured by interoperation and reuse, not by replacement. The same principle extends to neighbouring communities: the Trinity does not ask the Erlang community to adopt Prolog, or the Semantic Web community to abandon RDF. It asks whether there is value in a shared execution substrate for logic-based agents on the Web – and offers a concrete proposal for what that substrate might look like.

## 10.2 How to deal with community fragmentation

> The Web is more a social creation than a technical one. I designed it for a social effect – to help people work together – and not as a technical toy. The ultimate goal of the Web is to support and improve our weblike existence in the world.
>
> *Tim Berners-Lee, 2000*

In addition to systems fragmentation, the authors of *Fifty Years of Prolog* (FYPB) identify the fragmentation of the Prolog *community* as a significant concern for

both the standardization process and the long-term growth of the language. This social fragmentation mirrors the technical landscape: application developers and researchers congregate around specific systems (SWI-Prolog, Ciao, XSB, etc.), forming distinct sub-communities.

While these groups foster valuable local discussions, their isolation leads to duplicated efforts, a dilution of Prolog's collective voice in the wider technological discourse, and a slower dissemination of innovations. This disjointed state hampers collaboration and complicates efforts to maintain a unified language standard.

We argue that by utilizing the Prolog Trinity ecosystem, we can bridge these sub-communities through four key pillars: interaction, collaboration, learning, and shared identity.

### 10.2.1 Coming together through interaction

A programming language ecosystem, defined as a dynamic network of software, developers, and system implementers, relies on continuous interaction for its health and sustainability. The advent of the World Wide Web fundamentally transformed these ecosystems, shifting them from localized groups to interconnected global communities via platforms like GitHub and Stack Overflow. While the Prolog community has leveraged these tools to some extent, the authors of *Fifty Years of Prolog* highlight the absence of a strong, unified online presence compared to the vibrant communities surrounding languages like Python or JavaScript.

The "All Things Prolog" website, sponsored by the Association for Logic Programming, represents the first significant step toward a system-independent homepage.[3] However, to truly combat fragmentation, Prolog requires more than a static homepage; it needs a dynamic *portal*. We must distinguish between the *technical* Prolog Web – the infrastructure of nodes and agents – and the *social* Prolog Web – the network of users and developers. These two concepts should not exist in isolation. In the spirit of "We Are the Web," we envision a portal that integrates them, serving as the definitive gateway to the Prolog Trinity ecosystem.

Such a portal would be far more than a repository of links. It would serve as a living library, hosting comprehensive documentation and tutorials specifically designed to guide users through Web Prolog and agent building. Beyond static resources, it would function as a dynamic registry, providing a service directory of available public Prolog nodes and services, making the distributed nature of the ecosystem visible and accessible. To foster human connection, the portal would act as a community hub featuring forums, news, and collaborative tools, ensuring that the social layer is as robust as the technical one. Finally, it would host the playground (see Section 10.2.3), an interactive environment for instant experimentation, allowing visitors to write and run code without leaving the browser.

[3] `https://prolog-lang.org/`

The underlying philosophy is that technical interoperability can be a powerful catalyst for community cohesion. If we can make the many different Prolog systems talk to each other over the Web, there is hope that the Prolog sub-communities will start talking to each other too.

### 10.2.2 Coming together through collaboration and sharing

Collaboration is only made possible through interaction. To strengthen the community, we propose a culture of "dogfooding" – where the community actively uses the Prolog Web to build the Prolog Web. The most immediate early adopters should be Prolog users and implementers themselves.

This extends to the concept of *sharing*. Currently, we share code and ideas. The Prolog Trinity ecosystem enables us to share *computational resources* and *live processes*. Imagine a future where a developer can not only share a library on GitHub but also spawn a live demonstration agent on a public node, allowing others to interact with it immediately. This shifts the paradigm from social coding to *social computing*.

FYPB notes that the community lacks a standard package manager and that portability issues extend to libraries and tools such as testing and documentation infrastructure. This strengthens the case for unifying web-based environments at the node level: a shared, profile-governed Web Prolog node can provide a single distribution channel for code, a single execution target for teaching material, and a coherent place to attach tool support, while still allowing multiple underlying implementations.

### 10.2.3 Coming together through learning

> Ah, Prolog. Sometimes spectacularly smart, other times just as frustrating. You'll get astounding answers only if you know how to ask the question. Think *Rain Man*. I remember watching Raymond, the lead character, rattle off Sally Dibbs' phone number after reading a phone book the night before, without thinking about whether he should. With both Raymond and Prolog, I often find myself asking, in equal parts, "How did he know that?" and "How didn't he know that?" He's a fountain of knowledge, if you can only frame your questions in the right way.
>
> *Bruce Tate*

The future of any technology is inextricably linked to education. For Prolog to thrive and expand its reach, we must provide engaging, accessible, and modern learning experiences. The inherently interactive and distributed nature of the Prolog Trinity, particularly Web Prolog, offers an unprecedented opportunity to create "Prolog Playgrounds" that go far beyond traditional textbook exercises or standalone interpreters.

### Prolog playgrounds

Figure 10.3 shows three of the currently existing web-based Prolog playgrounds: SWISH (based on SWI-Prolog), the Ciao Prolog playground, and the Tau Prolog sandbox. These tools allow visitors to program in Prolog directly in the browser without installing software, and they have proven invaluable for education and quick prototyping.

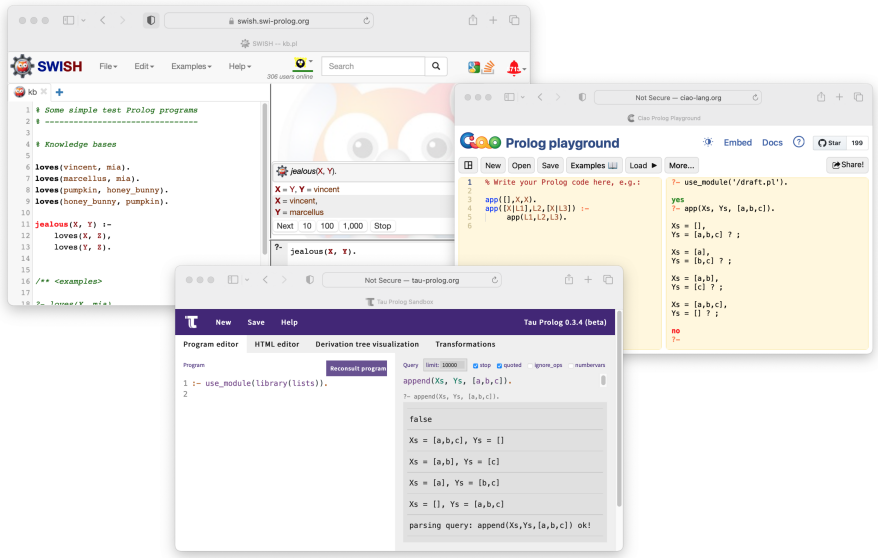



**Fig. 10.3** Current web-based environments: SWISH (SWI-Prolog), Ciao Prolog playground, and the Tau Prolog sandbox.

However, as things now stand, these playgrounds unfortunately mirror the very fragmentation of the community they serve. While they share the goal of bringing Prolog to the Web, they do not support the *same* language. Each playground runs a specific dialect – SWI, Ciao, or a JavaScript-based ISO subset – meaning that code written in one environment may not be portable to another without modification. A student or developer learning via SWISH encounters friction when moving to the Ciao playground, not only due to syntax divergences but also because the GUIs behave differently. The shortcuts, query mechanisms, and visualization tools are inconsistent, forcing users to relearn the tool rather than focusing on the logic. Furthermore, these distinct implementations represent a significant amount of duplicated engineering effort. Significant resources have been poured into developing three separate, siloed web interfaces – effort that could have been far more effective if directed towards a collaborative initiative.

Fortunately, it is not too late to change course. Building a unified Prolog playground served directly by a Web Prolog node is entirely feasible. An ISOTOPE node

would be fully capable of serving a standard Prolog environment suitable for general instruction. Moreover, if served by an ACTOR node, the environment could evolve into a playground for teaching Erlang-style actor programming in Web Prolog, empowering a new generation of developers to experiment with distributed concurrency within a single, coherent framework.

Prolog is indeed known to have a steep learning curve, and Erlang is probably not *that* far behind. Alas, there is nothing in Web Prolog that promises to make its learning curve less steep, but at least, to be sure, people already familiar with both Prolog and Erlang will have an easy task picking up Web Prolog, including the trickier areas such as concurrent programming. Prolog programmers without Erlang experience will be able to use Web Prolog out of the box – they only need to learn new things if they want to delve into concurrent programming in Web Prolog, and there are Erlang textbooks from which the principles as well as some of the practice can be learned. Erlang programmers not familiar with Prolog will have some new things to learn, but the close affinities between the two languages are likely to be of help here too.

By making the learning process more interactive, more contextualized within modern web technologies, and more directly connected to the exciting possibilities of distributed intelligent systems, these Prolog Playgrounds can ignite the curiosity of a new generation of programmers and researchers. They can showcase Prolog not as a historical artifact, but as a living, breathing language uniquely suited to addressing some of the most exciting challenges in computing today.

### Learning materials

Staying close to traditional Prolog also makes it possible to reuse some of the old textbooks written for teaching Prolog. There are plenty of really good Prolog textbooks around, and many can be downloaded for free. Some great books teaching Erlang exist too – and again, some of them are free.

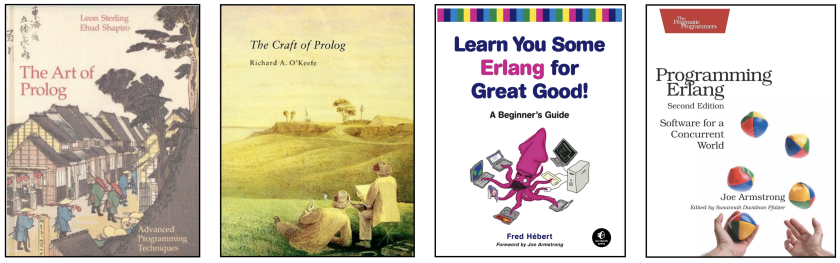



**Fig. 10.4** Textbooks for learning Prolog and Erlang – there are many others!

Learning concurrent and distributed programming in Web Prolog by reading Erlang books is surprisingly easy. Of course, at some point is would be wise to produce a dedicated book or at least a tutorial.

### 10.2.4 Coming together through rallying around a shared symbol

> One of the deep mysteries to me is our logo, the symbol of lust and knowledge, bitten into, all crossed with the colors of the rainbow in the wrong order. You couldn't dream a more appropriate logo: lust, knowledge, hope and anarchy
>
> *Jean-Louis Gassée*

A logotype gives a programming language a unique look and feel that makes it instantly recognizable and easier to remember. It is more than just a nice design – it is a way for users to connect, feel part of a community, and rally around a shared symbol. The logo also says something about what the language stands for, helping people get a quick sense of its vibe. Plus, it is useful for spreading the word; a good logo can make the language more visible in tutorials, online discussions, and conferences, making it stick in people's minds and building a legacy over time. Here are the logotypes for four well-established languages:


**Fig. 10.5** Logotypes for four programming languages.

By contrast, there is no Prolog logo broadly recognized across sub-communities. It may be that lack of a formal logo is intended to make a statement, signaling that Prolog is academic software? However, in recent years, even niche languages have started adopting logos as branding and community-building become more central, especially in open-source and web-facing contexts. We do believe it is time for Prolog to follow suit.

Fortunately, we need not begin from scratch. The field of logic programming already possesses an emblematic resource: the white-on-black geometric symbol at the heart of the logo for the *Association for Logic Programming* (ALP).


**Fig. 10.6** Logotype for the Association for Logic Programming.

Familiar to logicians, the three glyphs ⊨, ≡ and ⊢ stand for semantic entailment, logical equivalence and syntactic inference, respectively. Though underutilized and perhaps

underrecognized, the equation expresses a core thesis of logic programming: that syntax and semantics, process and meaning, are formally intertwined. The equation, while not fully realized in practical Prolog, remains a guiding principle of logic programming – and a fitting philosophical emblem for Prolog.

In the spirit of reuse and renewal, and adding a frame and a bit of color, we present the following proposal for a Prolog logotype:


**Fig. 10.7** A possible Prolog logotype.

By replacing the name "Prolog" with the names of other logic programming languages, and varying the color of the name and the frame, logotypes for other languages implementing this paradigm can be formed. Figure 10.8 shows four examples:


**Fig. 10.8** Logotypes for four other logic programming languages.

Of course, the logos need to look 2025-modern, not 1995-modern, so these visual mockups need to be professionally rendered if they are to be taken seriously.

Perception matters. A modern, compelling visual identity – a Prolog Web logotype – can serve as a powerful symbol of unity, innovation, and shared purpose. It helps to present Prolog not as a relic of the past, but as a forward-looking technology ready to tackle contemporary challenges. This is part of the "rebranding Prolog" effort that we believe is essential.

## 10.3 Extending Prolog's user base

> There are people who don't like popularity. It's much better to be exclusive and remote.
>
> *Robert Indiana*

### 10.3.1 The imperative for growth

Prolog *used* to be somewhat popular. In the eighties, during the second AI summer, the Japanese Fifth Generation Computer Systems (FGCS) project placed Prolog at

the center of the computing world. Communities formed, excellent textbooks were written, and the language became more efficient and expressive.

Today, the landscape is different. The authors of *Fifty Years of Prolog and Beyond* (FYPB) admit that Prolog has a "comparatively small user base." This is an understatement. Data presented by Statista estimates that in 2023 there were around 28 million programmers worldwide. By contrast, the SWI-Prolog forum – likely the largest gathering of Prolog developers – has only around 1,300 members. The Erlang forum has roughly 2,300. Compare this to Python, which boasts over 1,600 *local* user groups across 191 cities. As far as we are aware, there are no local Prolog user groups in any city.

The authors of FYPB identify this comparatively small user base as a significant threat. The Prolog community faces a crisis not of capability, but of invisibility. It is not just that Prolog is underrepresented in educational curricula or industry job postings. More critically, it lacks the mainstream discourse presence and cultural visibility that define a healthy language ecosystem. Few conferences center on Prolog, and few young programmers encounter it outside of narrow academic contexts.

Why does this matter? Prolog is an excellent language, but being good does not by itself make a language popular. In fact, the causality often works the other way around: a popular language is more likely to improve and become better. Popularity is not merely an intrinsic merit or a vanity metric; it is the engine of ecosystem health. Widespread use brings crucial benefits: more libraries, better tooling, more critical feedback, and greater pressure for innovation. Paul Graham articulated this dynamic in *Hackers & Painters* (2004):

> So whether or not a language has to be good to be popular, I think a language has to be popular to be good. And it has to stay popular to stay good. The state of the art in programming languages doesn't stand still. And yet the Lisps we have today are still pretty much what they had at MIT in the mid-1980s, because that's the last time Lisp had a sufficiently large and demanding user base.

While the Lisp ecosystem has since seen a revival through Clojure, Graham's central point holds true: a large user base fuels the ecosystem that, in turn, increases the utility and appeal of the language itself.

### 10.3.2 Key factors for popularity

Graham has more to say about popularity and what it takes for a programming language to become popular:[4]

> Programming languages don't exist in isolation. To hack is a transitive verb – hackers are usually hacking something – and in practice languages are judged relative to whatever they're used to hack. So if you want to design a popular language, you either have to supply more than a language, or you have to design your language to replace the scripting language of some existing system.

[4] `http://www.paulgraham.com/popular.html`

Applying this logic, supplying an entire web-based ecosystem, rather than just a language in isolation, presents a compelling path forward and offers a viable alternative that could replace JavaScript for some applications within this specific domain.

Antonio Cangiano, a software developer and technical evangelist at IBM, provides a more comprehensive list of key factors for popularity to consider:[5]

1. Being a default language for a popular ecosystem;
2. Being backed and promoted by a tech giant;
3. Having a large standard library and/or targeting a popular VM (e.g., JVM), unless it is a system language;
4. Fully embodying a new paradigm shift;
5. Being very useful in a particular domain, like Ruby did through Rails back in 2005;
6. Having great documentation, guidance for newcomers, tools, editor integration, and in general removing any instrumentation obstacle that makes getting started with your language difficult;
7. Fostering a welcoming community that encourages sharing, marketing, and evangelism;
8. Having a somewhat familiar syntax that is easy to read, write, and teach;
9. Being active and developed for several years;
10. Providing technical innovations that lead to productivity and more maintainable code;
11. Luck.

As Cangiano points out, the first two key factors are out of the question for most languages, and even then, few languages will meet all of the remaining requirements. Even embracing just a few of these should, however, be enough to grant a language a chance at becoming mainstream. So how well does Prolog satisfy these requirements, and how can Web Prolog help?

As for being the default language for a popular ecosystem (e.g. JavaScript for the Web), we are trying to *build* an ecosystem, namely the Prolog Trinity, for which Web Prolog is meant to be the default language. Whether the Prolog Trinity ecosystem will become popular remains of course to be seen. But we do not have to rely on the entire Prolog Trinity becoming a success. Even if the Prolog Web only finds few uses, Web Prolog as a special-purpose web programming profile of Prolog may still have an interesting role to play.

Being backed by a tech giant would certainly be nice, but being backed by the W3C might also help. If standardization is successful, industry backing and corporate sponsors that drives marketing or integration work might well show up.

Web Prolog does not target a specific VM. Any VM or any programming language that can possibly support an implementation of a profile of Web Prolog is targeted. After all, this is the whole idea behind a language aiming at real-time interoperability between different programming systems that may have different VMs or no VM at all.

---

[5] `http://programmingzen.com/so-you-want-your-programming-language-to-be-popular/`

As for fully embodying a new paradigm shift, we could possibly claim that the combination of web logic programming and web agent programming proposed in this book might be counted as a new paradigm, based on the two older paradigms of logic programming and actor programming. However, it may be more reasonable to think of Web Prolog as a multi-paradigm programming language.

Regarding the usefulness in a particular domain, creating a domain-specific profile of Prolog is our main focus – we aim to make Web Prolog very useful in the web domain. To aim for the kind of dominance Ruby on Rails had in this domain would be shooting for the stars. In addition, optionally in combination with SCXML, Web Prolog also tries to become a viable language for implementing sophisticated web-based agent programming platforms and intelligent conversational systems.

Documentation of the Web Prolog built-in predicates can be pooled from the documentation provided by existing Prolog systems. Guidance for newcomers in the form of textbooks, many of which are free, exists. As for removing other obstacles that make getting started with Web Prolog difficult, web-based IDEs such as SWISH must again be mentioned.

Currently, the most active Prolog community is arguably the one formed around SWI-Prolog. It is indeed a welcoming and encouraging community.

Whether or not having a somewhat familiar syntax that is easy to read, write and teach applies to Web Prolog depends on where you are coming from. For developers raised on Python or JavaScript the syntax of Prolog may seem alien at first, but for an Erlang programmer, Web Prolog will present few problems.

Regarding the factor of "having been active and developed for several years," we can of course not claim this for Web Prolog. However, the first Prolog system was developed already in the early seventies, and Erlang was conceived of and implemented during the late nineties, so from this perspective Web Prolog certainly has roots stretching deep into the history of programming languages, and through shared DNA with its parent languages a certain maturity of the underlying ideas can be claimed for it.

As for providing technical innovations leading to productivity and more maintainable code, this is what Prolog and technologies such as constraint logic programming is for.

And yes, we would certainly need luck as well, but as the Spanish proverb says, luck comes to those who look after it.

To these key factors, we are going to add another one, namely the existence of a standard. For a web technology such as Web Prolog, where interoperability between different implementations is at stake, standardization is particularly important.

Standardized languages tend to have larger communities and ecosystems surrounding them. This means there are more resources available, such as documentation, tutorials, libraries, and frameworks, which can accelerate development and provide solutions to common challenges. A robust community fosters collaboration, innovation, and knowledge sharing.

### 10.3.3 Rebranding Prolog

> *Rebranding* is a marketing strategy in which a new name, term, symbol, design, or combination thereof is created for an established brand with the intention of developing a new, differentiated identity in the minds of consumers, investors, competitors, and other stakeholders.
>
> *Wikipedia*

Here is an interesting quote from a Hacker News user that calls himself "gruseom:"[6],[7]

> With programming languages, I've come to the conclusion that "If X is so awesome, why is nobody using it?" is the wrong question to ask. Or rather, it's a fair question, but the answer doesn't depend on X. It is overwhelmingly a matter of network effects and social proof. What matters is what everyone else is using.
>
> A programming language has a window of opportunity to gain mindshare while it's new. Once that window passes, the odds are hugely against it. It's true there are a few technical factors, for example that older implementations lack the infrastructure to work with newer technologies, but such things can be improved. The dominant factor, the real barrier, is social psychology. This is obscured by our tendency to retrofit plausible-sounding reasons to our choices ("X is slow" or "it doesn't have libraries" or whatever).
>
> This does suggest a strategy for reviving an old language: rebranding. First, you need a new implementation. Trying to convince people to use the old one is like trying to get them to watch an old movie. George Lucas didn't promote Kurosawa, he made Star Wars. Second, there has to be a hook, something to convince us that this really is a new language, modern and with the times. So, light sabers. That's critical for opening a fresh window of opportunity instead of getting pegged to the old, long-closed one. It needn't be the most important thing, but it can't be fake either; it must be real enough or no one will take the new brand seriously. Without a convincing reason for why we weren't using X before, the new window will never open. Once it opens, then and only then do the beauty and power of X get their chance to bond with the user. In the Star Wars analogy, that would be the characters and the story – the deeper things that came from Kurosawa and ultimately from ancient archetypes. They'd never have gotten the chance to kick in if Star Wars hadn't got the audience in the theater. Once they did, though, then you had Star Wars fans for life.
>
> With Clojure, the hook was Java libraries. Actually, there were three hooks. The others were concurrency and sequences. But Java libraries is the one that worked. And now there's a modern Lisp again!

With Web Prolog, the hook is the Prolog Web with its focus on bringing more logic to the Web and the promise of real-time interoperability between different Prolog (and non-Prolog) systems. Not every new or rebranded programming language comes with a novel extension of the Web, so from a marketing perspective it would make sense to emphasize the Prolog Web as the "killer application" (or "killer middleware"?) for Web Prolog. Actually, there are three hooks here too. The other hooks are actors and a mature concurrency programming model with a proven track record thanks to Erlang and Elixir.

Since the author quoted above regards Clojure as a rebranded Lisp, and since it seems reasonable to regard Elixir as a rebranded Erlang, a comparison between the

---

[6] `https://news.ycombinator.com/item?id=3900047`

[7] His real name is Daniel Gackle and he is actually the head of Hacker News.

three "rebranding projects" may be interesting. Figure 10.9 is meant to illustrate such a comparison.

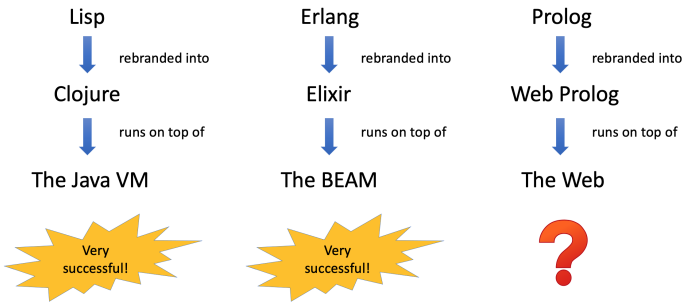



**Fig. 10.9** Lisp rebranded into Clojure and Erlang rebranded into Elixir, compared with Prolog rebranded into Web Prolog

Apart from the fact that we *know* that Lisp was successfully rebranded into Clojure, and Erlang into Elixir, but are less sure of how it would work out in the case of Web Prolog as a rebranded version of Prolog, there are other notable differences. In contrast to general-purpose languages such as Clojure end Elixir, Web Prolog is a *profile*, not a fork. For this reason, also in contrast to Clojure end Elixir, Web Prolog places great weight on backwards compatibility, i.e. on being able to run the vast majority of examples appearing in old but still very good Prolog textbooks. In other words, for Clojure and Elixir, the language has changed, but not its purpose. For Web Prolog, the language will stay (almost) the same, but its purpose has become a different one, namely web programming.

Moreover, Web Prolog does not run on top of a VM, but on top of the Web, propped up by nodes that serve as runtime systems disguised as web servers. This makes a lot of sense for a web programming language. Also, while the motive behind the rebranding of Lisp may well have been to create (what "gruseom" refers to as) a *modern* language, this is not the motivation behind Web Prolog. We are happy with "good ol' Prolog" adapted to the Web, and with a standard that would allow us to regard Web Prolog as a web technology in its own right.

The history of Elixir offers a specific lesson for us. Elixir did not become popular solely on its language merits; it exploded in popularity because of the Phoenix web framework. Phoenix provided a "killer app" environment – a high-performance, real-time web framework that made the power of the Erlang VM accessible and productive for web developers. The Prolog Trinity aims to create a similar effect. By providing a complete ecosystem – the Prolog agents and the Prolog Web architecture – we offer more than just a language; we offer a platform. Just as Phoenix made Elixir the go-to choice for scalable web apps, the Prolog Trinity can make Web Prolog the go-to choice for intelligent, distributed agents.

The ultimate purpose of this rebranding effort, and of the broader work of constructing an ecosystem around Web Prolog, is to set a virtuous cycle of adoption

in motion. As the community grows, the ecosystem becomes richer: more users generate more libraries, more tools, and more opportunities for integration. A larger user base also produces a broader and more accessible corpus of learning materials – tutorials, courses, examples, and best-practice guides – which directly addresses one of Prolog's long-standing challenges, namely its reputation for a steep learning curve. With broader engagement comes attention from industry, and with that attention follows investment, commercial experimentation, and a willingness to fund further research and development.

The notion of "rebranding Prolog" is central to this strategy. It is not about hiding Prolog's identity, but about reshaping its narrative. We are moving from a story about the past – expert systems and 1980s AI – to a story about the future: a web of intelligent, interoperable agents. Extending the user base is the only way to ensure that Prolog not only survives but thrives, contributing its unique logic programming paradigm to the next fifty years of computing.

## 10.4 Risk analysis

> I've heard that the Web is dead. But all the applications that have killed it are accessing services using HTTP over port 80.
>
> *Nat Torkington*[8]

In a thread on SWI-Prolog's Discourse forum addressing the "marketing" of Prolog, Paulo Moura voiced some scepticism regarding our approach:[9]

> As important as the Web is, and therefore the significance of Prolog being a good programming language for Web applications, the rebranding you suggest is overreaching and doesn't make sense to me. Prolog, and logic programming in general, is much more than a specific application field. It's already bad that, in some circles, Prolog is being downplayed as a DSL tool. The last thing we need is for Prolog to be downplayed as a Web thing. Prolog is a general programming language and that's the vision a lot of people in the community is working hard to establish. Trying to sell Prolog was a Web silver bullet would only backfire in the same way Prolog association with the hype surrounding AI backfired in the past. While the Web can be a stage to increase logic programming marketshare and mindshare, is not and should not be, in my view, the end game.

Upon reflection, describing Web Prolog as a "special-purpose language" was too narrow. Prolog should be recognised as a general-purpose language, while Web Prolog is better characterised as a specialised *profile* – a carefully delimited subset of the larger Prolog language. The right way to state the intent is therefore not that Prolog should be redefined as a web language, but that it should gain a clear web-facing profile and an interoperable surface. Prolog remains a general-purpose language; Web Prolog is a specialised profile that makes certain choices explicit at the boundary, so that nodes and clients can interoperate across the open network. The

[8] As quoted by Mike Loukides in ??, attributing the remark to Nat Torkington.

[9] `https://swi-prolog.discourse.group/t/encouraging-industry-about-prolog/1106/58`

Web, on this view, is not an end game or a silver bullet, but a particularly effective stage on which Prolog systems can be made discoverable, composable, and runnable by others.

The deeper point is that "web-facing" need not mean "web-only." Many general-purpose languages became widely used by first becoming indispensable in a particular arena: SQL in databases, JavaScript in browsers, R in statistics. A credible entry point is not a loss of generality; it is often how generality is earned.

The Web is a particularly plausible entry point for Prolog because many web problems have an implicit logical shape: query and transformation over structured data, policy and access control, constraint-based selection, and reasoning over graphs. Presenting Prolog as a rules-and-constraints component inside a polyglot web architecture is therefore not a marketing trick; it is an alignment between strengths and an enormous application surface.

The main failure mode to avoid is not "being on the Web," but making vague claims. The Trinity's stance is deliberately modest: specify clear interfaces, provide runnable nodes, and let adoption be driven by concrete utility rather than by rhetoric.

Wielemaker agreed with Moura, and followed up with the following statement:

> Prolog is quite suitable for the web, but in my experience Prolog is used for a very diverse set of tasks, many of which need Prolog more than the web. I'd love to see Web Prolog fly, in which case it is not unlikely to become more or less detached from the Prolog systems we see targeting other applications.

While we do agree that Prolog is used for many different task not related to the Web, we believe that there are ways to ensure that Web Prolog does not become detached from systems targeting other applications. The key here is to regard Web Prolog, not as a special-purpose language, but as a *profile* of the general-purpose Prolog language, and as a profile of a *future* Prolog. We refer to this future language as ISO Prolog 2.0.

We have a proposal for a path for Prolog that takes us from ISO Prolog to Web Prolog and then from Web Prolog to ISO Prolog 2.0. Such a path can be discerned against the backdrop of the profile diagram in Figure 10.10, which extends the diagram introduced in Chapter 3 with yet another profile, for ISO Prolog 2.0.

ISO Prolog 2.0 might include (at least) what Web Prolog did not include from ISO/IEC 13211-1:1995, such as predicates for file I/O. It might also include tabling, constraint logic programming or whatever technologies research into logic programming has come up with when the world is ready for it. Perhaps it might even be the Unified Prolog discussed earlier in this chapter.

Our proposal means, and it follows from the Venn diagram, that an implementation of an ISO Prolog 2.0 system would only be conformant if it implements the Erlang-style predicates for concurrency and distribution as well as the web APIs on which distribution depends. This might involve other forms of APIs as well (such as raw TCP/IP) and an extended set of predicate options, that allow Erlang-style distribution in a closed environment to work.

This suggests that a plan for the design and implementation of Web Prolog can easily be made to "fit inside" a plan for the design and implementation of ISO Prolog 2.0, one project inside another.

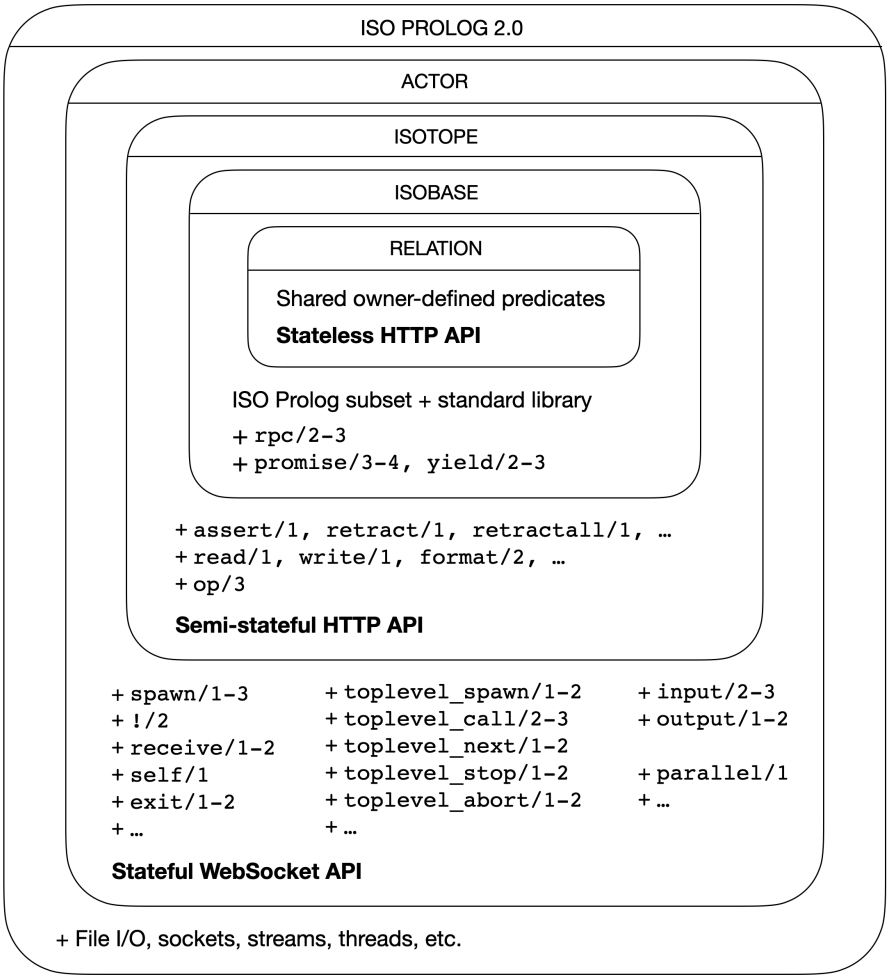



**Fig. 10.10** Web Prolog as a subset of a hypothetical future ISO Prolog 2.0.

It so happens that concurrency is included in the recipe for a Unified Prolog. Could the community be persuaded that Erlang-style concurrency is the way to go, then standardizing the relevant built-in predicates can be commenced already now, which basically means that the design and development of Web Prolog and of ISO Prolog 2.0 can be done in parallel.

One hope that we may have is that Web Prolog can inspire more and intensified work on the ISO Prolog standard as well as on the Prolog Commons. Perhaps the standardization of constraint logic programming is the next task for the ISO working group to tackle? Or perhaps finalizing DCG is more important.

# Chapter 11
# What's in it for other communities?

The long-term vitality and reach of the Prolog Trinity ecosystem depends on its ability to attract and serve a broader constellation of users beyond the traditional Prolog community. Within this ecosystem, logic programming researchers gain a venue in which their ideas can not only be demonstrated in a suitable GUI, but also *used* in practice through remote calls from external applications. Researchers and practitioners in the Semantic Web and Big Data worlds may find in Web Prolog a natural complement to existing technologies, since Prolog is well suited to data preprocessing, interactive construction of small, composable views, and transparent integration of heterogeneous sources such as relational databases. Symbolic and neuro-symbolic AI developers can exploit the Prolog Web's declarative expressiveness, agent-based communication model, and support for distributed inference, using Prolog both as a foundational language for symbolic AI and as a common representation for rules extracted from sub-symbolic predictors or for constraints imposed on machine-learning models to mitigate bias and opacity. Educators, system architects, and developers of intelligent web applications may likewise be drawn to an ecosystem that excels at building expert systems and logic solvers (probabilistic, abductive, or inductive) over heterogeneous data, particularly as it matures and integrates more tightly with established Web protocols and standards. In this way, dogfooding by the core community not only strengthens the platform from within but also demonstrates its readiness to welcome and support a diverse range of users.

Figure 11.1 introduces symbols that hint at neighbouring ecosystems. The emblem with nested red and black circles at node `n4.org` is the logo for *cplint*, a probabilistic logic programming platform. The blue database cylinders represent RDF triple stores, standing in for Semantic Web back-ends that can be exposed as Prolog Web nodes. On the right, the familiar Python logo appears together with the OpenAI emblem, indicating a Python-based service that fronts a large language model such as ChatGPT and makes it available to other agents via a FFI.

TODO: Caption must be fixed

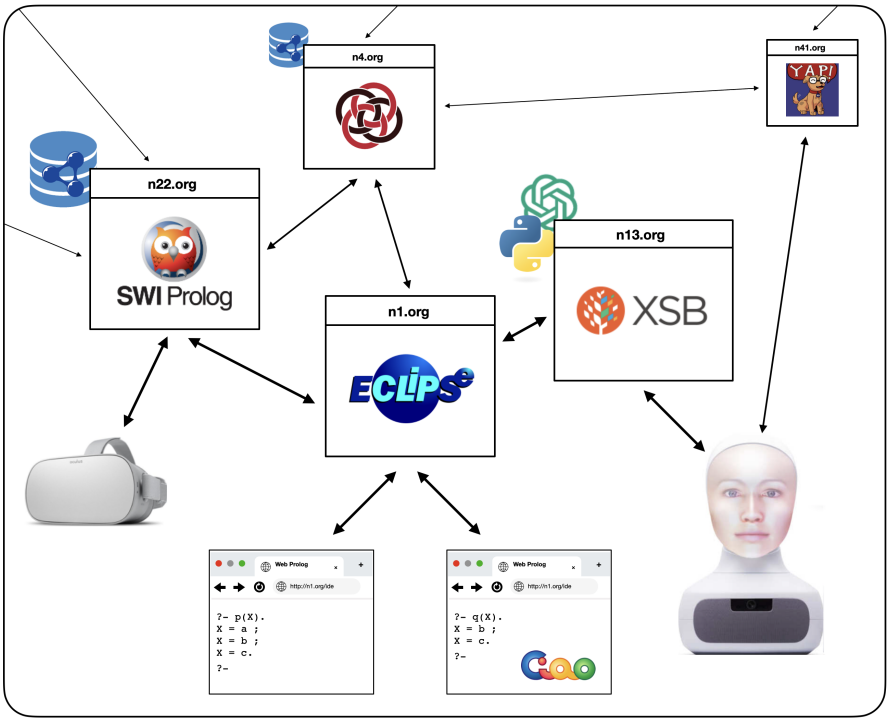



**Fig. 11.1** The Prolog Trinity: End-user clients in interaction with Prolog agents on the Prolog Web. (The logotype with nested red and black circles is a system for probabilistic logic programming.)

These additions suggest that Prolog nodes are intended to live in the same networked space as Semantic Web infrastructure and neural/ML services. Section 11.2, 11.3 and 11.4 return to these connections from the perspectives of the Logic Programming, the Semantic Web and AI communities respectively. What front-end programmers may gain is discussed in Section 11.5. Last but not least, although Erlang is not explicity represented in Figure 11.1, it is covered in Section 11.6.

## 11.1 For the broader logic programming community?

Earlier chapters have treated the Prolog Web primarily as a response to the current state of Prolog and its ecosystem, and as a concrete way of putting the Prolog Trinity into practice. In this section we take a broader view and ask what the proposal might offer to the wider logic programming community, including traditions that do not identify themselves as "Prolog systems" in any narrow sense. If you work on

Answer Set Programming, Datalog variants, constraint systems, probabilistic logic programming, or other LP formalisms, what is in it for you?

The short answer is: a Web-native integration layer that your system can join with minimal effort. Section 2.1 surveyed the diversity of the logic programming landscape; Section **??** argued that the Prolog Web's profile hierarchy can accommodate that diversity without forcing any system to adopt features foreign to its paradigm. Here we demonstrate this concretely.

As illustrated in Figure **??**, several research systems can already be deployed on the Prolog Web. The integration paths differ in detail but share a common pattern: wrap the system's query interface in the Web Prolog protocol, advertise the appropriate profile, and participate.

Consider probabilistic logic programming. The cplint system, introduced in Section 2.1, is implemented as a SWI-Prolog library and is already accessible through "cplint on SWISH." Because SWISH builds on Pengines, cplint is in practice already reachable through the Pengines protocol – and therefore through the subset of Web Prolog that Pengines implements. The following example illustrates this directly.

### Pengines calling cplint

Suppose you have two urns: urn1 contains 40 blue balls and 20 red balls and urn2 contains 25 blue balls and 30 red balls. You throw an unbiased coin and, if it turns out head, you draw a ball from the first urn, if it turns out tails you draw a ball from the second urn. Write a program modeling this process and a query for answering the question "What is the probability of drawing a blue ball?"

```
?- pengine_rpc('http://cplint.ml.unife.it', prob(blue,Prob), [
     src_text("
       :- use_module(library(pita)).
       :- pita.
       :- begin_lpad.
          blue : 2/3; red : 1/3 :- urn1.
          blue : 25/55; red : 30/55 :- urn2.
          urn1 : 0.5 ; urn2 : 0.5.
       :- end_lpad.
     ")
   ]).
Prob = 0.5606060606060606.
?-
```

LPS, also implemented in SWI-Prolog and SWISH, can be accessed in the same manner. However, as it now stands, the result we receive is a large term (a dict) that serves as instructions for how to display the result graphically.

ProbLog presents a different integration path. As discussed in Section 2.1, ProbLog 2 is a Python-based toolbox with a strictly declarative execution model:

it does not support side-effecting operations such as I/O or dynamic modification of the clause database. Rather than being a limitation, this makes ProbLog an ideal candidate for an ISOBASE node, since ISOBASE assumes nothing more than a pure ISO-style toplevel without mutable state or Erlang-style concurrency. Wrapping the ProbLog runtime as an ISOBASE node would allow other agents to submit goals, obtain answers, and incorporate probabilistic reasoning results into larger workflows on the Prolog Web.

Multi-paradigm languages such as Picat offer yet another integration path. Picat's constraint solving, planning, and optimisation capabilities could be exposed as specialised services on the network, addressable by any client that can issue a Web Prolog query.

The common pattern is that participation does not require rewriting a system from the ground up. It requires a thin protocol layer – a wrapper that translates between the Web Prolog message format and the system's native query interface – and a declaration of the appropriate profile. Systems built on SWI-Prolog are closest to immediate deployment; systems with their own runtimes need a lightweight adapter, but the protocol is deliberately simple enough to make this practical.

The broader point is architectural. The logic programming community has produced a remarkable diversity of formalisms and implementations, but they remain largely isolated from one another. A researcher using ProbLog cannot easily delegate a constraint satisfaction sub-problem to a Picat node, or combine probabilistic inference with ASP-style model generation, without building bespoke integration code. The Prolog Web offers a common substrate – a shared protocol, a shared addressing scheme, and a shared notion of "ask a question, get answers" – that could turn isolated systems into a federated network. Whether such a Logic Programming Web will emerge depends on adoption by these communities, but the infrastructure described in this book is designed to make that adoption as frictionless as possible.

## 11.2 For the broader logic programming community? - OLD

Earlier chapters have treated the Prolog Web primarily as a response to the current state of Prolog and its ecosystem, and as a concrete way of putting the Prolog Trinity into practice. In this section we take a broader view and ask what the proposal might offer to the wider logic programming (LP) community, including traditions that do not identify themselves as "Prolog systems" in any narrow sense. If you work on Answer Set Programming, Datalog variants, constraint systems, probabilistic logic programming, or other LP formalisms, what is in it for you?

The architectural vision behind Web Prolog and the Prolog Web is not limited to the Prolog tradition. It points toward the possibility of a Logic Programming Web in a broader sense.

Most logic programming systems take advantage of the unique Prolog features. LPS, already discussed in Chapter 7, certainly does.

As illustrated in Figure 11.1, research areas such as probabilistic logic programming can already be deployed on the Prolog Web: systems such as *cplint*, *ProbLog*, and multiparadigm languages such as *Picat* can be wrapped in a Web Prolog–compatible interface and immediately participate. These examples demonstrate that the Prolog Web need not be confined to classical Prolog, but can serve as an integrating substrate for a wide range of logic programming technologies.

- **cplint** is a suite of tools for probabilistic logic programming, implemented as a Prolog library and available in particular for SWI-Prolog. It supports both inference and learning for Logic Programs with Annotated Disjunctions (LPADs) and related formalisms under the distribution semantics, allowing users to define discrete probability distributions and continuous probability densities directly in Prolog. cplint is accessible both as a local Prolog pack and through "cplint on SWISH," a web-based environment where probabilistic logic programs can be edited, queried, and learned from data via a browser.
- **ProbLog** is a probabilistic logic programming language that extends Prolog with probabilistic facts, allowing each fact to hold with a given probability and thereby combining logical atoms with random variables under the distribution semantics. A ProbLog program defines a probability distribution over possible worlds, and queries return the probability that a given atom is true rather than just a yes/no answer. The current ProbLog 2 system is a Python-based toolbox offering efficient algorithms for probabilistic inference, most probable explanations, sampling, and parameter learning, and is available both as a library and via online tools from the DTAI group at KU Leuven.
- **Picat** is a multi-paradigm, Prolog-like logic-based programming language that integrates logic programming, constraint solving, dynamic programming, planning, and scripting in a single framework. It offers several specialized solver modules and a high-level modeling style where problems are expressed declaratively and solved via built-in search and optimization mechanisms. Picat's pattern-matching rules, tabling, and constraint-based primitives make it well suited for combinatorial search, AI planning, and optimization tasks, while its familiar syntax and tight integration of different paradigms aim to provide a practical alternative to both traditional Prolog systems and dedicated constraint or planning languages.

## Pengines calling cplint

Suppose you have two urns: urn1 contains 40 blue balls and 20 red balls and urn2 contains 25 blue balls and 30 red balls. You throw an unbiased coin and, if it turns out head, you draw a ball from the first urn, if it turns out tails you draw a ball from the second urn. Write a program modeling this process and a query for answering the question "What is the probability of drawing a blue ball?"

```
?- pengine_rpc('http://cplint.ml.unife.it', prob(blue,Prob), [
      src_text("
```

```
            :- use_module(library(pita)).
            :- pita.

            :- begin_lpad.

            blue : 2/3; red : 1/3 :- urn1.
            blue : 25/55; red : 30/55 :- urn2.

            urn1 : 0.5 ; urn2 : 0.5.

            :- end_lpad.
        ")
   ]).
Prob = 0.5606060606060606.
?-
```

LPS, also implemented in SWI-Prolog and SWISH, can be accessed in the same manner. However, as it now stands, the result we receive is a big term (an dict) that serve as instructions for how to display the result graphically.

A key enabler of this integration is the node profile architecture introduced in earlier chapters. Since not all systems support the same operational features, the profiles ensure that each system participates at a level consistent with its expressive capabilities. This becomes particularly clear in the case of *ProbLog*. Designed around the distribution semantics, ProbLog maintains a strictly declarative execution model and does not support side-effecting operations such as I/O or dynamic modification of the clause database. Rather than being a limitation, this makes ProbLog an ideal candidate for an ISOBASE node, since ISOBASE assumes nothing more than a pure ISO-style toplevel without mutable state or Erlang-style concurrency. Wrapping the ProbLog runtime as an ISOBASE node allows other agents to submit goals, obtain answers, and incorporate probabilistic reasoning results into larger workflows on the Prolog Web. More feature-rich systems may adopt the ISOTOPE or ACTOR profiles, but languages such as ProbLog fit naturally and cleanly into the ISOBASE profile.

A major design motive is precisely this: to enable existing Prolog systems – diverse in capabilities, internal representations, and implementation philosophies – to communicate with one another using Web Prolog as a *lingua franca*. By presenting each system as a node that exposes a uniform Web Prolog interface, heterogeneous engines can exchange data, script one another's toplevels and actors, and cooperate in distributed computations. Extending this principle, any system for which such an interface can be provided – deductive databases, non-Prolog logic programming languages, or systems written in Erlang, Java, JavaScript, or Python – can join the same conversational fabric by adopting the profile appropriate to its operational model.

From the perspective of the broader logic programming community, the Prolog Trinity ecosystem thus offers a concrete path toward greater interoperability and visibility. Over the years LP has developed a rich variety of languages and reasoning

paradigms, including Answer Set Programming, Datalog variants, constraint systems, and probabilistic frameworks. Yet these systems often remain isolated, each with its own tools, formats, and runtimes. A logic-enabled Web of communicating agents provides a unifying context in which these approaches can interact without sacrificing their distinct strengths. One might imagine Web Prolog agents consulting ASP solvers for combinatorial tasks, invoking Datalog engines over large datasets, or delegating probabilistic inference to specialised systems such as ProbLog or *cplint* – each wrapped as a Prolog Web node under the profile suitable to its capabilities.

This book does not attempt to contribute to the logical foundations of logic programming itself; instead it focuses on extra-logical infrastructure: the practical mechanisms that allow diverse systems to coexist, interoperate, and be deployed on the Web. In that sense, the Trinity is not merely a proposal for the Prolog community narrowly conceived. It is an invitation to the entire logic programming field to anchor its research prototypes, experimental systems, and mature engines in a shared, web-native environment, where ideas can circulate more freely, capabilities can be reused across implementations, and logic programming can present a more coherent face to the wider world.

## 11.3 For the semantic web community?

Chapter 5 argued that Web Prolog and the Prolog Web can be positioned as a kind of *semantic-web logic programming language* and a complementary "third tower" alongside RDF, OWL, and SPARQL. Rather than rehearse those arguments, this section looks at the same picture from the point of view of researchers and practitioners already committed to the Semantic Web stack. The question is simple: if you work with RDF graphs, OWL ontologies, and SPARQL endpoints, what is in it for you?

A first answer is that the Prolog Trinity ecosystem offers a missing *programming layer* for the Semantic Web – a layer where logic and action are integrated, and where semantic applications can not only query and infer, but also react, maintain state, and communicate. RDF, OWL, and SPARQL provide a powerful foundation for representing and querying knowledge, but they stop short of providing a uniform, web-native language in which semantic agents can be written. Web Prolog fills precisely that gap. It gives Semantic Web developers a logic-based scripting language in which they can express rules, policies, and workflows that live *on top of* RDF data and ontologies, instead of having to embed such logic in ad-hoc JavaScript, Java, or Python code around a triple store.

A second answer is that the Trinity supplies a concrete and deployable *agent model* for the Semantic Web. Chapter 7 introduced the Agentic Web as a vision of a Web populated not only by linked datasets but by software agents capable of perceiving, reasoning, and acting. The Prolog Web realises a fragment of that vision: Web Prolog agents can consume RDF data, consult OWL ontologies, perform symbolic inference, and then publish new knowledge or trigger follow-on actions. For Semantic

Web practitioners who have long spoken about "semantic agents" in rather abstract terms, the Trinity offers an explicit architecture and a programmable substrate for building them.

Third, the Prolog Web gives the Semantic Web community a practical way to connect heterogeneous reasoning systems without abandoning the existing stack. Prolog nodes may be backed by persistent RDF triple stores, relational databases, graph databases, or specialised reasoners such as probabilistic or abductive engines. From the outside, these appear as Prolog Web nodes with appropriate profiles, exposing logical APIs that exchange ISO-compatible terms.

Seen from this perspective, Web Prolog is not a rival to RDF, OWL, or SPARQL, but a way of harvesting decades of logic-programming research to make the Semantic Web more usable. Features such as tabling, constraint handling rules, and well-founded semantics already exist in mature Prolog systems; the Prolog Trinity ecosystem proposes to reuse this infrastructure rather than re-implement it in bespoke rule engines attached to triple stores. For Semantic Web practitioners, this means that advanced reasoning facilities – non-monotonic reasoning, or domain-specific rule languages – can be prototyped and deployed on top of existing data.

The architectural side of the story is equally important. The Prolog Web provides a disciplined way of deploying such logic-based components as nodes in a distributed system that speaks standard Web protocols. Nodes can front-end SPARQL endpoints, serve or consume RDF, or participate in Linked Data workflows, while also hosting agents that communicate asynchronously and carry out higher-level tasks. This suggests a path toward what one might call a *Semantic Prolog Web*: a hybrid in which RDF and OWL continue to carry the semantic load, while Web Prolog and Prolog agents provide the programmable glue, orchestration, and long-lived behaviour that many applications need.

Finally, there is an institutional and standardisation angle. The W3C Semantic Web activity has produced an impressive stack of standards for data and ontology representation, but has so far remained largely programming-language neutral. The Trinity proposal invites the Semantic Web community to consider whether it might be time to recognise a web-native logic programming language, developed in concert with the ISO Prolog community, as a first-class tool in its ecosystem. Supporting the development and standardisation of Web Prolog, the Prolog Web, and Prolog agents would not replace the existing stack; it would supply a coherent way of turning its machine-understandable semantics into running, distributed, semantically aware applications.

If that invitation is accepted, the benefits are mutual. The Semantic Web gains a practical, logically grounded programming and agent layer; the Prolog Trinity gains a natural home in an established standards landscape. Together they move the Web one step closer to a world in which linked data is not only published and queried, but also interpreted and acted upon by a rich population of logic-based web agents.

## 11.4 For the AI community?

Chapter 7 painted a broad picture of Prolog AI agents on the Agentic Web: symbolic reasoning engines encapsulated as actors, distributed across the network, orchestrating both other symbolic services and neural components such as large language models. In this section we revisit that picture from the vantage point of the wider AI community. If you work on machine learning, symbolic AI, multi-agent systems, or conversational agents, what does the Prolog Trinity ecosystem offer you?

Figure 11.1 provides a visual starting point. The central nodes in the diagram host different Prolog systems, but they are surrounded by symbols that clearly point beyond classical logic programming: on the left, a VR headset suggests immersive user interfaces in which Prolog agents act as the cognitive core behind virtual or augmented experiences; on the right, a social robot indicates embodied agents capable of speech, gaze, and gesture, driven by logic-based controllers deployed on the Prolog Web. Near the top, the Python logo together with the OpenAI mark represents a Python-based service that fronts a large language model such as ChatGPT and exposes it as a web API. Web Prolog actors can send prompts to such a service, receive responses, and integrate them with other symbolic knowledge. The same infrastructure can connect to other machine-learning services, data stores, and external tools. The message of the figure is that Prolog agents are meant to inhabit a mixed ecosystem in which traditional AI components, neural models, and interactive front-ends all communicate through a common web-native fabric.

A first benefit for the AI community is that the Prolog Web offers a web-native arena for AI systems. Modern AI already lives on the Web: models are trained on web-scale data, deployed behind web APIs, and consumed through browsers, mobile apps, and embedded devices. The Prolog Web takes this seriously and proposes a programming model in which AI components are not isolated binaries hidden behind opaque endpoints, but named, message-receiving actors inhabiting a shared, logically structured space. For AI researchers interested in reproducible experiments, deployable prototypes, or long-lived services, this means that symbolic components, decision procedures, and coordination mechanisms can be deployed as first-class web entities rather than as ad hoc infrastructure.

Second, the Trinity provides a mature substrate for symbolic and explainable AI. Chapter 7 revisited Prolog's classic strengths in knowledge representation, constraint solving, planning, meta-interpretation, and proof construction. Those strengths become particularly valuable in a world where black-box models dominate: Web Prolog agents can maintain explicit knowledge bases, construct and expose proof trees, encode norms and policies, and mediate between human-understandable rules and data-driven predictions. For AI practitioners facing regulatory or ethical demands for transparency, this makes Web Prolog a serious candidate for the reasoning core of systems that must explain themselves, justify their actions, or enforce formal constraints.

Third, the Prolog Web supports a principled approach to agent-based AI. Toplevels and other actors already form a simple multi-agent substrate; on top of that, more sophisticated agent architectures (BDI-style agents, SCXML statechart actors, con-

versational web agents) can be built in a language that directly supports logical inference and structured communication. User-interface agents – embodied conversational agents, dialogue systems, assistants – can be implemented as Web Prolog actors that combine DCG-based natural language processing with actor-style interaction. Multi-agent-system (MAS) agents can negotiate, allocate resources, and coordinate plans across multiple nodes using the same message-passing primitives. The result is an agent framework in which both UI-agents and MAS-agents can be expressed using a common logical substrate.

At the same time, much remains open at the level of agent abstractions. Chapter 7 also surveyed a number of existing Prolog-derived agent frameworks, such as DALI and LPS, and argued that Web Prolog is a natural host for similar ideas on the Agentic Web. What is still missing in the Trinity ecosystem is a small, well-defined agent framework that is specific to Web Prolog and lives on top of the existing actor and node architecture. Such a framework would have to identify a modest set of concepts – events, goals, plans, beliefs, norms, perhaps a restricted form of mental state – and map them cleanly onto ACTOR nodes and their mailboxes, without undermining the simplicity of the underlying concurrency model. It should also take seriously the lessons from previous agent languages and from current initiatives such as the W3C Autonomous Agents on the Web Community Group, which promotes hypermedia MAS aligned with the Web Architecture. From an AI perspective, designing, implementing, and evaluating such a Web-Prolog-native agent layer is an important piece of future work, because it would provide a more user-friendly programming model for building Prolog agents on the Agentic Web and a concrete artefact around which the Prolog, Semantic Web, and AI communities could coordinate their efforts.

Fourth, the Trinity suggests a concrete path toward neuro-symbolic systems. Rather than trying to implement deep learning inside Prolog, the proposal is to treat neural components – large language models, classifiers, embedding services, vision models – as callable services that can serve as the definitions of selected predicates. Web Prolog programs then orchestrate these services, enforce symbolic constraints, maintain state, and integrate results with other sources of knowledge. This division of labour respects the strengths of each paradigm: neural models specialise in pattern recognition and language modelling; logic programs specialise in structure, constraint, and explanation. Because LLMs and other neural models are typically exposed via web APIs, they fit naturally into the Prolog Web's communication model, as illustrated by the Python and OpenAI logos in Figure 11.1.

There is also a historical lesson. Expert systems struggled in part because there was no natural infrastructure on which to deploy them, and because knowledge acquisition was labour-intensive. The Web changes both conditions. A web-based agent infrastructure gives symbolic systems a natural deployment platform, and LLM-based tools can ease some aspects of knowledge acquisition, for example by helping to draft rules or transform natural-language descriptions into formal representations that can then be curated and constrained by human experts. In such architectures, Web Prolog can act as the backbone that connects symbolic rules, machine-learned models, and human input into a coherent agentic whole.

Interestingly, there is a very active W3C Community Group dedicated to development of theories and practices for the advancement of more sophisticated web agents.[1] Here is how they describe their interests and goals:

> **AUTONOMOUS AGENTS ON THE WEB COMMUNITY GROUP**
> This community group is interested in the design of Web-based Multi-Agent Systems (MAS) for the deployment of world-wide hybrid communities of people and artificial agents on the Web. Our aim is to design a new class of Web-based MAS that are aligned with the Web Architecture to inherit the properties of the Web (world-wide, open, long-lived, etc.), and are also transparent and accountable to support acceptance by people. We are especially interested in the use of Linked Data and Semantic Web standards for weaving a hypermedia fabric that enables uniform interaction among heterogeneous entities: people, artificial agents, devices, digital services, knowledge repositories, etc. We refer to this new class of Web-based MAS as Hypermedia MAS (hMAS). This community group brings together experts actively contributing to advances in autonomous agents and MAS, the Web Architecture and the Web of Things, Semantic Web and Linked Data, and Web standards in general – as well as any other areas that could contribute to this approach for distributed intelligence on the Web.

It seems to us that the Prolog Trinity ecosystem might be an ideal foundation for building communities of such agents. Its actors are explicitly aligned with the Web Architecture, communicate through standard web protocols, and can be equipped with autonomy in the sense required by Hypermedia MAS: they can follow links, interpret representations, maintain internal state, coordinate with other agents, and respect declaratively specified norms and policies. For the AI community, this means that the abstractions proposed by the W3C group are not merely aspirational. With Web Prolog as the agent language, the Prolog Web as the substrate, and Prolog agents as the computational entities, it becomes possible to experiment with web-scale autonomous agents on a platform that is both logically grounded and architecturally compatible with the Web itself.

For researchers and practitioners in AI, then, what is in it is the opportunity to develop and deploy symbolic, agent-based, and neuro-symbolic systems on a principled web-native foundation. Web Prolog will not replace Python, TensorFlow, or PyTorch, but it can stand alongside them as the logical control layer that ties components together, reasons about their behaviour, and exposes their actions in a form that both humans and other agents can understand.

—

Furhat is a tabletop social robot head developed by the Swedish company Furhat Robotics, designed as a highly expressive platform for face-to-face spoken interaction.[2] The robot combines back-projected facial animation onto a physical mask with motorised head movements, microphones, speakers and cameras, yielding an embodied conversational agent whose identity (face, voice, personality) can be reconfigured in software. Furhat's software stack includes both a local "skill" framework and a realtime WebSocket (WS) API that exposes the robot as a networked resource: external programs written in any langue can connect over WS to send utterances and control non-verbal behaviour, and to receive events such as user speech, gaze and presence.

---

[1] `https://www.w3.org/community/webagents/`

[2] I am not affiliated with Furhat Robotics, but I know some members of the team personally.

Over this realtime interface it would be straightforward to control every aspect of the conversational behaviour of a Furhat robot from an actor running on an ACTOR node, or from a statechart actor coordinating complex multimodal behaviour, thereby treating the robot as a remote embodied agent within the Prolog Web. As a result, Furhat has become a versatile research platform in human-robot interaction, social robotics and embodied conversational agents, supporting studies in areas such as turn-taking, multi-party interaction, social signalling, education, and healthcare scenarios. The company offers a freely downloadable SDK with a "virtual Furhat" for development and research prototyping, while physical robots and advanced cloud services are provided under commercial licences and subscription packages targeted at universities, labs and enterprise deployments.

—

The W3C Machine Learning Working Group [1] focuses on the Web Neural Network API (WebNN) [2] along with an informative report on ethical principles for Web machine learning [3].

WebNN provides a platform independent API for executing artificial neural network models. More specifically, inference not training, despite the name of the Working Group. The API is currently a W3C Candidate Recommendation. You can try it out on Chrome browsers, but will first need to enable this with a browser flag.

Despite the name of the Working Group, WebNN is actually targeted at inference, not training. Unlike PyTorch, it omits support for back propagation and stochastic gradient descent. The existing libraries (Tensorflow and ONNX) are huge, and moreover, lack a high level representation for defining neural network models.

## 11.5 For web programmers and the future of web development?

The vast and diverse community of web programmers – working daily with JavaScript and a wide range of server-side languages and frameworks – forms much of the backbone of today's digital world. For this community, the Prolog Trinity – and Web Prolog in particular – offers a genuinely different way of thinking about certain classes of web applications and services. It is not a call to abandon existing stacks, but an invitation to treat logic as a first-class instrument in web development when the problem domain calls for it.

From a web programmer's perspective, one of the most immediate attractions of Web Prolog is its suitability for rule-intensive applications. Contemporary web stacks are remarkably versatile, yet they can become cumbersome when handling intricate business rules, layered validation logic, sophisticated configuration schemes, or highly personalised behaviour based on many interacting conditions. In such cases, large bodies of imperative code are often needed to simulate what is, at heart, a set of logical constraints and decision rules. Web Prolog offers a more direct route: developers can encode this structure as logical relations and rules, and rely on the inference engine – with unification and backtracking – to explore the relevant

cases. For certain classes of problems, the resulting code can be more concise, more transparent, and easier to adapt as requirements evolve.

The Trinity also encourages a shift in how we think about the dynamics of web applications. Much of the traditional Web still operates in a client-driven request-response style: a browser or mobile app asks, a server answers. In contrast, the Trinity's long-lived, communicative agents are designed to live continuously on the network. They can subscribe to feeds, monitor data streams, react to events, initiate contact with other services, and cooperate to achieve goals. In the terminology of this book, they inhabit an *Agentic Web*: a web of durable participants and ongoing interactions rather than a web of one-shot calls. For web programmers, this suggests a move towards more proactive, event-driven, and intelligent entities, where some application behaviour is delegated to long-lived agents instead of being recomputed from scratch on each request.

Data integration is another area where Web Prolog can play a useful role. Modern web applications routinely draw on many APIs and heterogeneous data sources, creating ad hoc "mashups" by stitching together HTTP requests, JSON transformations, and local caches. Web Prolog, with its relational core and strong support for querying and transforming structured data, offers a natural environment for building more principled integration services. Information from multiple origins can be represented in a uniform logical form, combined using rules, and subjected to explicit constraints. This is also where alignment with RDF-style representations becomes practically useful. The aim is not to replace existing APIs, but to provide a logic-based layer in which their results can be related, checked, and reasoned about.

None of this removes the fact that Prolog has a learning curve, especially for programmers who have spent their careers in imperative or object-oriented paradigms. Unification and backtracking are conceptually different from variables and loops, and it would be naive to pretend otherwise. The Prolog Trinity therefore tries to make the transition as concrete and rewarding as possible. The focus is on web-centric use cases where the benefits of logic programming become visible quickly; on clear, documented APIs that hide some of the low-level details; and on educational resources and playgrounds aimed specifically at web developers, as discussed in earlier chapters. The claim is that, for the right kinds of tasks, the time invested in learning Web Prolog can be repaid in clarity, robustness, and speed of development.

A related theme is the need to make Prolog legible again to the modern developer. The point is not to disguise its identity, but to move beyond the dated picture of Prolog as a niche language from the expert-systems era. The Trinity presents Prolog instead as a language well suited to contemporary concerns: concurrency, distributed agent architectures, web-native communication, and logical intelligence at the edge of applications. For web programmers, the message is that Prolog is no longer confined to standalone AI prototypes or academic examples; it can participate directly in the infrastructure of real web systems.

## Constructing user interfaces with statecharts

One area in which the Trinity's ideas connect directly to a recognised pain point in web development is user-interface state management. Web applications must ensure that users can perform only the operations that are valid at a given moment, that asynchronous events do not leave the interface in an inconsistent state, and that edge cases – a button pressed twice, a network timeout during a form submission, a navigation event during a pending request – are handled gracefully rather than ignored. These are control problems, and they grow harder as applications become more interactive.

The JavaScript ecosystem has responded with a succession of state-management libraries – Redux, MobX, Vuex, Zustand, and others – each offering a different trade-off between explicitness and convenience. These tools have improved the situation considerably, but they address the problem at the library level rather than at the level of the underlying model. The developer still reasons about state transitions implicitly, as scattered conditional logic embedded in event handlers and reducers.

Statecharts offer a more structured alternative. As developed in Chapter 6, a statechart makes control explicit: the set of possible states, the events that trigger transitions, the guards that constrain them, and the actions that accompany them are all declared as inspectable structure rather than buried in procedural code. This is not a new idea in the web world. In his book *Constructing the User Interface with Statecharts*, Horrocks (1999) argued for statecharts as a design tool for GUIs. More recently, David Khourshid's XState library[3] has brought SCXML-compatible statecharts to the JavaScript front-end community, and his conference talks on the subject have resonated with developers frustrated by ad hoc state management.[4] David Junger (2014) was an early proponent of applying statecharts to web interfaces.

The core insight behind this movement can be stated concisely. A user interface is a reactive system: given a current state and an incoming event, it must produce a new state and a set of actions. A statechart formalises exactly this relationship, decomposing the state space into a hierarchy of named states with explicit transitions, and thereby making the control structure of the interface visible, testable, and communicable to non-programmers. The advantages – visual design, exhaustive enumeration of modes, decoupling of the state machine from the rendering code – are precisely what web developers need when applications grow beyond a handful of interactive elements.

Where the Trinity adds something to this picture is in the combination of statecharts with a logic-based scripting and data-modelling language. In the XState world, executable content is JavaScript: guards are functions, actions are functions, and the data model is whatever JavaScript data structures the developer chooses. In the Trinity's statechart actors, the corresponding role is played by Web Prolog. Guard conditions become Prolog goals that succeed or fail; actions become Prolog predicates that can query a knowledge base, enforce constraints, construct structured

[3] `https://github.com/statelyai/xstate`

[4] See also the resources collected at `https://statecharts.github.io/resources.html`.

messages, or invoke remote services; and the data model inherits Prolog's relational and symbolic strengths. For applications in which the interface must reason – must check policies, validate configurations, maintain and query a structured model of the domain – this combination is more natural than embedding such logic in JavaScript callbacks.

The practical question is whether web developers would adopt such an approach. The answer depends in part on the availability of Web Prolog in browser environments, a topic discussed in Section **??** and acknowledged in Chapter 12 as one of the main open engineering challenges. But it also depends on whether the web development community continues to recognise state management as a fundamental difficulty rather than a solved problem. The evidence from the steady growth of interest in XState and related tools suggests that it does. If statecharts are indeed the right abstraction for governing complex interactive behaviour, then the question of what scripting language sits inside them becomes practically important – and Prolog's case for that role, in domains where logic and constraint matter, is strong.

More broadly, the relevance of statecharts to web development reinforces a point made throughout this book: that the distinction between "front-end" and "back-end" obscures a shared underlying problem. Whether one is managing dialogue turns in a voice interface, protocol states in a distributed agent, or navigation flows in a single-page application, the challenge is the same: to govern reactive behaviour so that it remains correct, inspectable, and maintainable as complexity grows. The statechart actor, as developed in Chapter 6, is the Trinity's answer to that challenge, and the web front-end is one of the most natural arenas in which to test it.

## 11.6 For the Erlang community?

Figure 11.1 does not display an Erlang logo, yet Erlang is present throughout the entire diagram in a less visible but more fundamental way: without Erlang's intellectual contribution, none of the arrows in the figure would mean what they do. The very possibility of Web Prolog agents communicating asynchronously, isolating their internal state, and coordinating distributed computations derives directly from ideas developed in the Erlang/OTP world. Readers from the Erlang community will recognise the design lineage immediately. What, then, might the Prolog Trinity ecosystem offer in return?

A first answer is that Web Prolog represents a re-engagement with a line of thought that once connected the two communities closely. Erlang's early prototypes grew out of experiments with Prolog, but in order to meet the stringent requirements of fault-tolerant telecom systems, the language shed much of Prolog's logical machinery – backtracking, unification-heavy control, meta-programming – in favour of determinism, reliability, and simplicity. The result was a brilliantly focused platform for soft real-time systems. Web Prolog now attempts the complementary move: to reintroduce rich logical capabilities into an actor-style framework that still respects the principles that made Erlang successful.

For Erlang developers, this offers a practical gain. Many applications built atop OTP involve increasingly complex decision-making logic, rich policy constraints, structured plans, or rule-governed behaviour. Encoding such logic in Erlang's functional idiom is certainly possible, but often verbose or difficult to validate. Web Prolog provides an alternative: the "brains" of an Erlang-style actor can be written declaratively, using unification, backtracking search, DCGs, constraint solvers, or rule interpreters. In other words, Erlang processes may remain responsible for everything they excel at – supervision, I/O management, binary protocol handling, predictable latency – while Web Prolog agents handle symbolic reasoning, explanation, and sophisticated decision procedures.

A second answer concerns interoperability and hybrid architectures. The Prolog Web aims to be a network of services that speak the same actor-oriented protocol. There is no reason an Erlang node could not sit in this network. Indeed, the BEAM VM remains one of the most robust actor engines ever built, and Erlang systems could participate as *first-class* nodes in a heterogeneous set of communicating services. They could delegate logical tasks to Web Prolog agents while providing high-performance endpoints for streaming, front-end connectivity, or massive connection handling. Conversely, Web Prolog agents could be orchestrated or supervised by Erlang systems familiar with OTP's reliability patterns. A shared communication fabric opens the possibility of hybrid systems that neither community has fully explored.

Third, Web Prolog offers a conceptual bridge. If Erlang brought message passing, isolation, linking, and supervision to the world of functional programming, the Prolog Trinity offers Erlang developers a chance to experiment with logical tools that can simplify agent complexity: constraint propagation, rule languages, proof trees, meta-interpreters, planning, explanation, and more. These features can complement, rather than compete with, Erlang's strengths. In this respect, the Trinity plays a role somewhat analogous to Akka for Scala or Cloud Haskell for Haskell – projects that sought to borrow the essence of Erlang's concurrency model for languages with richer type systems or evaluation strategies. The difference is that Web Prolog was designed from the outset with actor semantics in mind, and its logical features sit *beside* rather than *on top of* the actor substrate.

Finally, there is a research opportunity. Several commentators have noted that while Erlang originally grew out of logic programming ideas, its present form is only one point in a larger design space. Web Prolog explores another. If one were to imagine a future Web Prolog VM that matched the responsiveness and low-latency scheduling of BEAM while still supporting unification, guarded backtracking, and meta-level execution, this would require new research into virtual machine architecture for logic-based actors. That research would draw naturally on decades of BEAM expertise. Conversely, Erlang developers interested in broadening the conceptual reach of Erlang-style concurrency may find in Web Prolog a living laboratory for ideas that cannot easily be explored inside the BEAM constraints.

In short, the Erlang community has already gifted the wider world a model of concurrency that is elegant, principled, and battle-tested. The Prolog Trinity ecosystem acknowledges that debt and offers something in return: a way to explore

richer logic inside an actor discipline, a platform for hybrid systems that combine robust process management with deep symbolic reasoning, and a research frontier where Prolog's declarative core and Erlang's architectural clarity meet again as peers. If successful, this could reopen a line of communication between two communities that once shared foundational ideas – and may once again have much to offer each other.

# Part VI
# Final words

# Chapter 12
# From vision to execution

> Vision without execution is just hallucination.
>
> *Thomas Alva Edison*

At the beginning of this book, the Prolog Trinity ecosystem was introduced by means of a mindmap. Its purpose was primarily orienting: to sketch the conceptual territory, to name its principal components, and to hint at the relationships between them. That diagram accompanied the reader through the early stages of the argument, serving as a visual shorthand for a still-emerging design.

In Figure 12.1 we now return to the same mindmap – structurally unchanged, but this time adorned with three banners of the kind commonly used by the W3C.

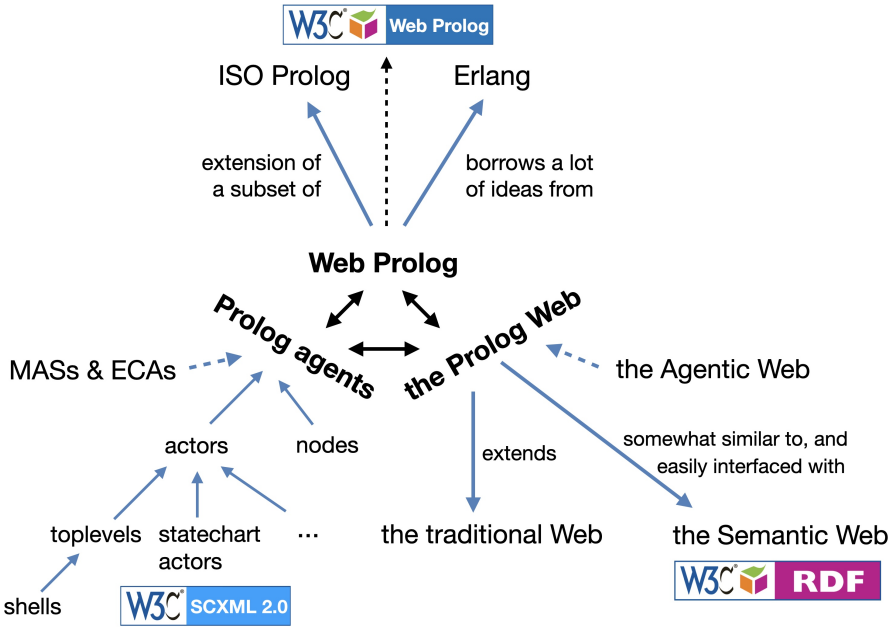



**Fig. 12.1** The Prolog Trinity mind map revisited, now framed as shared infrastructure and adorned with three W3C-style technology banners.

Such banners are not mere decoration. They function as compact visual markers of collective agreement: signals that a technology has passed from individual proposal to shared standard, and that its definition, maintenance, and evolution are now matters of public process rather than private intent. In a single glyph, a banner compresses years of discussion, specification work, experimentation, revision, and implementation experience.

Of the three standards implicitly invoked by the Prolog Trinity vision, only one currently carries such a banner: RDF. Web Prolog and the statechart language do not. This asymmetry is neither accidental nor embarrassing. On the contrary, it clarifies the status of the Trinity today. RDF represents a language whose role in the Web ecosystem has already been executed in the strongest possible sense: specified, standardised, implemented, deployed, criticised, and revised. Web Prolog and the statechart language, by contrast, remain parts of a forward-looking design – carefully argued, technically grounded, but not yet socially ratified.

This chapter is concerned with what it would mean to close that gap. The chapter therefore has two distinct but closely related tasks. First, it revisits the vision itself, drawing together the main ideas developed throughout the book and presenting them once more as a coherent whole. Second, it considers the work required to move from vision to execution: not only the technical work of implementation, but also the social and institutional work through which languages become shared infrastructure. The banners in Figure 12.1 should thus be read not as claims of authority, but as markers of intent – placeholders for standards that do not yet exist, but whose contours this book has attempted to make precise.

## 12.1 The vision

> Vision without action is a daydream. Action without vision is a nightmare.
>
> *Japanese proverb*

> In order to carry a positive action we must develop here a positive vision.
>
> *Dalai Lama*

This section is written at a higher altitude than most of the book. The goal is not to introduce new machinery, but to restate the proposal as a coherent vision.

### 12.1.1 Agents as the primary abstraction

The Trinity's most important structuring commitment is to treat agents as the primary abstraction: long-lived processes with identities, mailboxes, lifecycles, and

explicit interaction protocols, hosted on nodes that themselves act as accountable participants rather than as anonymous servers. Chapter 9 develops this stance in detail, including the architectural multi-agent reading and the way it connects profiles, capability policy, supervision, and the sequential/concurrent partition to application pressures such as interactive control, coordination, and recovery. Here we simply recall the consequence: once "agents all the way down" is taken seriously, the Prolog Web becomes not only a network of endpoints, but a programmable society of communicating entities.

### 12.1.2 Executable logic at web scale

The Trinity's claim that logic on the Web should be treated as *executable* rather than merely representational is developed in Chapter 9. The execution task, in the remainder of this chapter, is to make that substrate real: specified at the boundaries, implemented end-to-end, and adopted as shared infrastructure.

### 12.1.3 Two partitions and the axes of design

The conceptual distinction between the sequential and concurrent partitions, together with the cross-cutting axes that shape design choices – purity and the openness-versus-risk trade-off – are developed in Chapter 9. Here we take those commitments as given. The execution question is how to turn them into deployable reality: which profiles to standardise first, which capability boundaries to make testable, and which security and governance mechanisms must be engineered before openness can be offered responsibly.

### 12.1.4 Webizing Prolog, prologizing the Web

In Section 1.6 we introduced a dual process which has guided the entire design: *webizing Prolog* and *prologizing the Web*. The first asks what happens when a language and its execution model are treated as part of an open world, and it yields a set of requirements that force Prolog to become web-native in its surface forms, its deployment posture, and its operational discipline. The second asks what it would mean for the Web to host, not only documents and services, but executable logic in the form of long-lived agents, and it yields a set of requirements that force the Web-facing substrate to preserve Prolog's distinctive strengths rather than merely embedding it as a curiosity.

This subsection returns to the concrete checklist stated in Section 1.6 and assesses, point by point, what has been proposed or demonstrated in the rest of the book. The

purpose is not to declare victory, but to make the scope and the remaining gaps explicit before we turn, in the next section, from vision to execution.

**Webizing Prolog: eight requirements.**

Section 1.6.1 listed eight requirements for webizing Prolog. We revisit them here in the same order.

1. *Introduce URIs into the ecosystem.* This requirement is addressed by treating naming and addressing as web-facing from the outset. In Chapter 4, node addresses are URIs or NodeIDs, actor identifiers are understood as locatable in a network of nodes, and remote interaction primitives such as `rpc/2-3` and the node option in `spawn/3` are designed to be used across node boundaries without changing the programming model. In other words, URIs are not bolted on as an I/O convention; they are part of the semantic surface by which a computation reaches beyond its local context.
2. *Exploit the existing web infrastructure.* Chapters 3–4 develop the Prolog Web as an explicitly web-aligned substrate: a stateless HTTP interface where it suffices, a stateful mode where interaction needs continuity, and a WebSocket mode where message ordering and long-lived sessions are operationally central. The JSON encoding of Prolog terms, along with the framing of request–response versus mailbox-driven interaction, is meant to allow ordinary web clients to participate without learning a bespoke transport discipline.
3. *Make use of existing means for security.* The design responds to the open-world setting by insisting on containment-by-default rather than assuming trust. Chapter 4 introduces the core mechanisms that make openness manageable: capability policy (including disciplined use of communication capabilities), an explicit owner–client split, and resource bounds that let a node remain usable under adversarial or simply careless workloads. That said, this is also the place where the book is most candidly incomplete. The architecture identifies what must be controlled and where the boundary should sit, but a robust, standardisable security story demands more engineering detail than has been supplied so far, including stronger guidance on authentication, authorisation, auditing, and sandboxing across different deployment environments.
4. *Ensure web-size scalability.* The scalability story in Chapter 4 is fundamentally distribution-first. The Prolog Web treats distribution as the normal case: nodes are autonomous, composition happens across nodes, and caching and load distribution are expected in the stateless HTTP mode. Session affinity, stateless caching, and the discipline of designing the sequential partition so it can be served by ordinary web infrastructure are all part of the attempt to scale in the way the Web scales: by replication, caching, and loosely coupled components rather than by a single globally coordinated runtime.
5. *Support web application development.* Chapter 10 sketches what it would mean for the Trinity to be a practical development platform rather than a pure research

proposal: turn-key nodes, playgrounds, and statechart-based approaches to user-interface state management. In this picture, a node is not merely a remote Prolog engine; it is a web-facing component that can be spun up, configured, and composed by non-specialists, and statecharts provide a principled bridge between reactive UI logic and Prolog-based modelling and scripting.

6. *Ensure fitness to web culture.* A web-facing system does not live only by its semantics. Chapter 9 therefore treats openness, documentation, community building, and the presence of a portal or hub as part of the design, not merely as marketing. This requirement is partly cultural and partly infrastructural: an open ecosystem needs canonical entry points, discoverable artefacts, and a public working style that matches how the Web community tends to build shared standards and shared libraries.
7. *Aim for standardisation.* Standardisation is separated out as an explicit execution task further ahead in this chapter, because the argument here is not that the book has already standardised anything, but that it has tried to identify a scope that could plausibly be standardised: small boundaries, precise interfaces, and a minimal interoperable core. The important claim at the vision level is that a web-facing Prolog substrate must be specified at the boundaries if it is to be shared, even if implementations remain free to innovate behind those boundaries.
8. *Play nicely with existing web standards.* This requirement is addressed across multiple chapters by treating Web Prolog as adjacent to, not opposed to, established W3C and IETF practices. Chapter 4 grounds the concrete transport and representation choices in the ordinary web stack (HTTP, WebSocket, JSON), keeping the system legible to web engineers. Chapter 6 connects to SCXML as a statechart language, suggesting a path by which statechart actors can align with existing state-machine standards. Chapter 7 relates the Prolog Web to the Semantic Web, emphasising compatibility with existing data standards rather than inventing new ones.

**Prologizing the Web: three requirements.**

Section 1.6.2 listed three requirements for prologizing the Web. These are less about adopting web infrastructure and more about ensuring that what becomes web-native remains recognisably Prolog and recognisably agentic.

1. *Leverage Prolog's capabilities.* Chapter 8 is the main place where the book argues that Prolog brings a distinctive set of capabilities to the Agentic Web layer: knowledge representation, inference, planning, meta-interpreters, DCGs, and the broader tradition of expert systems. The point is not nostalgia, but reuse: these are mature, well-understood techniques that become newly relevant when deployed as long-lived, tool-using agents embedded in a communicative network.
2. *Leverage Erlang-style concurrency.* Chapters 2–5 develop the actor model strand of Web Prolog: processes with mailboxes, message ordering as a semantic concern, failure propagation, supervision, and reusable actor patterns captured

as generics. This is the prologizing half of the argument in its most concrete form: the Web is not merely a place to call a Prolog predicate, but a place where Prolog processes can live, coordinate, and recover under explicit operational rules.

3. *Run in browsers.* Section 4.3.2 sketches a browser profile and discusses WebAssembly as a possible substrate. This is the other major gap, alongside security. The book motivates why browser execution would be powerful and how it could fit the profile hierarchy, but it does not yet offer a fully worked operational account of scheduling, isolation, shared state, and performance in a browser-hosted ACTOR environment. The requirement is therefore acknowledged as only partially addressed: the direction is clear, but the engineering is not yet complete.

**An honest assessment.**

Taken together, the checklist suggests that the Trinity has made substantial progress on the architectural and interface-facing requirements of webizing Prolog: URIs and web addressing, alignment with HTTP and WebSocket interaction styles, JSON-based interchange, and the explicit separation of sequential request–response computation from concurrent mailbox-driven interaction. The proposal has also made concrete progress on the prologizing requirements that concern expressive power and concurrency: Chapter 8 makes the case for Prolog's symbolic strengths in the Agentic Web setting, and Chapters 2–5 develop an actor model that is strong enough to support long-lived interaction and supervision.

The main open gaps are also clear. First, security is architecturally well covered in Chapter 4: the open-world posture is grounded in three structural mechanisms – inexpressibility, invisibility, and containment – complemented by standard web deployment practices. The remaining execution work is to make this security posture operationally concrete: profiles must be accompanied by conformance tests, and public-node deployments need a baseline recipe for authentication, quotas, auditing, and transport security.

Second, browser execution is motivated and sketched, but not yet backed by a detailed, testable account of how the ACTOR profile behaves under the constraints of browser runtimes. These two gaps are not peripheral. They are precisely the points where an appealing model meets the open-world reality of the Web. For that reason, they reappear as central execution tracks in the remainder of this chapter.

### 12.1.5 Statechart actors and the governance of interaction

Statechart actors, and the broader claim that complex systems require explicit *governance of interaction*, are developed as part of the Trinity's paradigmatic stance in Chapter 9 (Section 9.6). Here, the execution question is how to make that stance

operational: which control notation to standardise or treat as interchange, how to test conformance, and how to engineer tooling that makes control structure inspectable in deployed systems.

### 12.1.6 A platform for the Agentic Web

Agentic-system pressures are a major part of what makes execution urgent: once systems consist of long-lived, tool-using processes operating over the network, questions of governance, failure handling, timing, and interoperability stop being optional. The work in this part of the book is therefore motivated not only by elegance, but by the practical demands of building agents that remain durably interactive under explicit rules in an open Web setting.

### 12.1.7 Federation, not unification

A central choice underlying the Trinity is to pursue federation rather than unification. The aim is not to design a new Prolog that all existing systems must become, nor to declare a single implementation as the reference centre of gravity. Instead, the proposal seeks to create shared *interaction surfaces*: web-facing interfaces and execution profiles that multiple systems can implement while remaining free to diverge internally.

This federative stance responds directly to the historical reality of the Prolog ecosystem. Prolog has thrived as a family of systems with different strengths, but that diversity has also produced a persistent portability and community-cohesion problem. The Trinity offers a different axis of standardisation: rather than standardising everything *inside* a system, standardise what must be shared *between* systems when they interact over the Web. In other words, federation treats interoperability as a public good and internal design freedom as a local right.

This is also why Chapter 10 and Chapter 11 belong to the conceptual core of the book rather than to its marketing layer. The claim is that a federated Prolog Web can change the community's practical incentives: when systems can meet through web-native APIs, shared portals, playgrounds, and demonstrators become meaningful, and progress can be measured by interoperation rather than by persuasion. Federation thus becomes a strategy for ecosystem repair: it creates a way to cooperate without demanding agreement on everything.

### 12.1.8 Prolog as a web technology in its own right

If the preceding subsections are taken seriously, the Trinity's rhetorical punchline is not merely that "Prolog can be used on the Web". It is that Prolog can be a web technology in its own right: a web-native medium for executable knowledge and interaction protocols, deployed as services and agents across an open network.

This reframing matters because it changes what counts as success. The goal is not only that a Prolog system can expose an HTTP endpoint. The goal is that Prolog becomes legible, usable, and adoptable in the way other web technologies are: with shared interfaces, test suites, playgrounds, public nodes, and a culture of composition. If that happens, Prolog's traditional strengths – explicit representation, rule-based inference, nondeterminism, and meta-programming – become available in the places where web applications and agentic systems increasingly need them.

Seen this way, the Trinity is not a bid to redefine Prolog, but a bid to give it a clearer web-facing posture. Prolog remains a general-purpose language; Web Prolog is a specialised profile whose main purpose is to make Prolog's symbolic strengths – explicit rules and auditable control – directly usable as an interoperable layer in agentic systems on the Web. The claim is not that the Web is the end game, but that it is a particularly effective stage on which heterogeneous systems can be composed and made trustworthy through explicit, inspectable interfaces.

Taken together, the preceding sections articulate a vision that is internally coherent and deliberately constrained. The Prolog Trinity has been presented not as a monolithic system, but as an ecosystem structured around clear roles, layered abstractions, and a principled division of responsibilities. At this level, the question is no longer whether the design hangs together – it does – but what it would mean to take responsibility for making it real.

The remainder of this chapter therefore shifts perspective. Having restated the vision one last time, we now turn to execution: to the technical, organisational, and communal work through which designs cease to be proposals and become shared infrastructure.

## 12.2 The execution

> We are not analyzing a world, we are building it. We are not experimental philosophers, we are philosophical engineers.
>
> *Tim Berners-Lee*

### 12.2.1 Standardization and shared ownership

#### Is it not too early?

At what point in the evolution of a programming language should one try to create a standard for it? Is it really reasonable to aim for a Web Prolog standard already at this point? Is it not too early? Perhaps the language needs to mature first. Perhaps someone needs to implement a stable, speedy, and secure node before we decide what should be normative. After all, that is the usual approach.

A first reply is that the ingredients are not new. Prolog and Erlang are mature languages that have stood the tests of time, and the Web standards on which this proposal relies are likewise well established. Web Prolog should therefore be seen as a product of incremental evolution rather than as a revolution: the task is not to invent an unfamiliar paradigm, but to find a disciplined way to combine existing ones.

A second reply is more important. It is indeed too early to standardize everything. Much of what makes the Trinity interesting still belongs to design exploration and engineering iteration. But it is not too early to standardize the shared surfaces that prevent fragmentation: the media type, the profile boundaries, the client to node contracts, and the conformance tests by which independent implementations can agree on what it means to interoperate. If these boundaries are left implicit for too long, they will be standardized anyway, but accidentally, by whichever implementations happen to dominate first.

In that sense, the question is not whether standardization should happen now, but what should be standardized now. The claim of this book is deliberately modest: standardize the interoperable core early enough to enable a shared ecosystem, while leaving room for implementations to compete, experiment, and mature behind those shared interfaces.

#### Who should do it?

If Web Prolog is to be both web-native and language-precise, it is difficult to see how a credible standardization effort could be carried out within a single culture of standardization. In our view, the W3C should not do it alone, and nor should ISO. The natural model is a liaison: a division of labour in which language-level concerns and web-architecture concerns are both represented as first-class, and where review across that boundary is routine rather than exceptional.

Liaisons between standards bodies are not controversial. The W3C engages in liaisons with numerous organizations,[1] and ISO likewise maintains liaison relationships that include W3C.[2]

---

[1] `https://www.w3.org/liaisons/`

[2] `https://www.iso.org/organization/10143.html`


**Fig. 12.2** Turning the Trinity design into shared infrastructure requires a liaison.

It would be premature to speculate about the exact form such cooperation might take for Web Prolog. The important point at this stage is simply that governance must match scope: language profiles, web protocols, and ecosystem-level interoperability are unlikely to be served well by treating the effort as belonging wholly to either camp.

### How do we do it?

We must not underestimate the difficulties – standardization is hard and often frustrating – but W3C provides a practical point of entry in the form of *W3C Community Groups*.[3]

> A W3C Community Group is an open forum, without fees, where Web developers and other stakeholders develop specifications, hold discussions, develop test suites, and connect with W3C's international community of Web experts.

The basic idea is that an individual can share a proposal, gather collaborators, develop a draft and a test suite, and publish a report that may later motivate the formation of a formal working group. At the time of writing, there are many such groups, spanning a wide range of topics.

Standardization is often misunderstood as a purely technical activity – the act of writing down interfaces, fixing semantics, and publishing specifications. While these steps are necessary, they are not sufficient. What distinguishes a standard from a well-documented design is not precision alone, but shared ownership.

Technologies that eventually acquire banners do so only after their authors have relinquished a certain degree of control. Decisions become matters of public record; trade-offs are negotiated rather than declared; and authority shifts from individual insight to collective process. In this sense, standardization is as much a social transition as a technical one.

[3] `https://www.w3.org/community`

**Where do we start?**

As we have argued throughout the book, the appropriate unit of standardization is not merely a language, but an ecosystem. Web Prolog is one essential strand, but the Prolog Web architecture, the profile hierarchy, and the externally observable behaviour of Prolog agents such as nodes and toplevels must also be specified. A viable standard must cover not only computation – syntax and semantics of primitives – but also the pragmatics of communication: protocols, APIs, message formats, and conformance conditions.

The good news is that much of the language-level ground has already been covered by the ISO Prolog standard. This means that the distinctive effort proposed here can focus on what is currently missing from the Prolog ecosystem: shared web-facing interaction surfaces and the contracts that make a distributed Prolog Web interoperable. The aim is not to freeze innovation, but to give it a stable substrate on which multiple implementations can meet.

### 12.2.2 Implementation roadmap

If standardization is the process by which a design becomes shared, implementation is the process by which it becomes real. No amount of conceptual clarity can substitute for running systems, and no specification can be considered mature until it has survived contact with actual use. For an ecosystem such as the Prolog Trinity, execution therefore requires a sustained commitment to implementation – not as a final validation step, but as a primary source of design feedback.

An implementation roadmap need not be ambitious in scope to be effective. On the contrary, early implementations are most valuable when they are modest, even mundane. Minimal nodes, incomplete shells, narrowly scoped agents, and unglamorous tooling are not signs of immaturity, but essential probes into the design space. They expose ambiguities in semantics, reveal hidden coupling between components, and force decisions that can otherwise remain comfortably abstract. In this sense, prototypes are not preliminary versions of a finished system, but instruments for discovering what a finished system could reasonably be.

For Web Prolog, this implies prioritizing executable semantics over completeness: small interpreters, reference runtimes, and test suites that make the language's behaviour observable and comparable across implementations. For the Prolog Web, it implies early experimentation with concrete protocols, failure modes, and resource constraints, even if these experiments only cover fragments of the envisioned architecture. Partial implementations that work end-to-end are more informative than comprehensive designs that remain untested.

Equally important is the role of redundancy. Multiple, independently developed implementations – even if limited and imperfect – provide a form of validation that no single reference system can offer. Divergences between implementations are not merely bugs to be fixed, but signals that a specification is underspecified,

overconstrained, or poorly understood. An implementation roadmap should therefore encourage diversity of approach rather than premature convergence.

Seen in this light, execution is less about marching toward a predefined endpoint than about maintaining a feedback loop between design and practice. The roadmap outlined here is intentionally open-ended: its purpose is not to predict how the Prolog Trinity ecosystem will be built, but to ensure that whatever is built remains grounded in experience rather than aspiration.

A first Web Prolog node, in this spirit, would likely be unremarkable in almost every respect. It might support only a subset of the language, expose a minimal HTTP-based query interface, and host a small collection of static programs and data. Concurrency could be limited, error handling crude, and performance uneven. Yet even such a modest system would already exercise the core ideas of the Trinity: executable logic served over the Web, queries whose behaviour can be observed and reproduced, and agents or clients interacting with a node through well-defined but still evolving contracts. By existing at all, such a node would force abstract notions – of purity, state, resource bounds, and interaction – to acquire concrete meaning, and would provide a shared point of reference around which further experimentation could take place.

### 12.2.3 Community and adoption

The preceding subsections have treated execution as a procedural and technical project: articulate an idea, publish a document, iterate through criticism and implementation, and arrive at a shared test suite with at least two interoperable implementations. That path is necessary, but it is not sufficient. A specification can be perfectly written and still inert. For the Trinity to become more than a private design, it must be adopted, used, and improved by a community that finds it worth building against.

Chapter 9 argued that the Prolog community's most persistent problem is not a lack of ideas but a lack of shared *interaction surfaces*. Fragmentation is sustained by the fact that systems, tools, teaching material, and sub-communities often do not meet in a common place. The Prolog Trinity ecosystem is therefore as much a community proposal as a technical one: technical interoperability is treated as a catalyst for social cohesion. If the systems can talk to each other over the Web, there is at least a chance that the communities will too. This is the motivation behind distinguishing between a *technical Prolog Web* (nodes and agents) and a *social Prolog Web* (people and collaboration), and insisting that they should be integrated rather than isolated.

A first practical step is the *portal* proposed in Chapter 9: a single public gateway that serves simultaneously as documentation hub, service directory, and community meeting place. The portal is not merely a set of links. It is meant to make the distributed nature of the ecosystem visible: which public nodes exist, what profiles they implement, what services they host, and what interaction protocols they expose. It is also meant to make the social layer visible: discussion forums, news, shared projects, and collaborative tools that invite participation from both users and implementers.

A second step is to treat *playgrounds* as first-class infrastructure rather than as isolated curiosities. The Prolog world already has web-based playgrounds, but they are fragmented by implementation and by interface conventions. A Trinity-oriented playground can be more than a browser REPL: it can be a profile-governed execution target for teaching material, examples, and small applications, and it can serve as the default way to try out remote queries, pagination-as-backtracking, and basic agent interaction without local installation. In other words, the playground becomes the point where the "run this now" culture of the Web meets the semantics of Prolog.

A third step is *dogfooding*: using the Prolog Web to build the Prolog Web. This is the most credible early-adoption story because it starts with the people who already care. If Prolog implementers and advanced users begin to publish nodes, services, and live demonstrations that others can interact with immediately, the ecosystem shifts from *social coding* to *social computing*. Sharing no longer means only pushing source code to a repository; it also means hosting runnable agents and queryable knowledge bases in a way that others can test, stress, and reuse. This style of participation is also valuable for standardisation, because it forces interface choices to survive contact with reality.

A fourth step is *learning material* that meets people where they are. One advantage of the Trinity approach is that it does not require abandoning existing Prolog culture. Tutorials and courses can still draw on established Prolog and Erlang textbooks, but they can be reframed around web-native interaction: "call this node", "compose these services", "spawn this agent", "observe this protocol". The result is a learning path that produces immediately usable skills in a modern setting, which is exactly what the Prolog community needs if it wants to attract new users without first demanding that they share decades of context.

Finally, adoption depends on allies beyond the Prolog community. Neighbouring communities provide both early adopters and complementary expertise. Logic programming researchers can help clarify semantics and constraints. The Semantic Web community can supply stable representations and tooling that the Trinity can consume and produce. The AI and agentic-systems community can stress-test the proposal against real conversational and tool-using agents. The Erlang community provides a mature culture of fault-tolerant concurrency, supervision, and operational discipline. Web programmers, in turn, provide the pragmatic sensibility that keeps interfaces and deployment habits aligned with web reality. In short, the Trinity is not only a technical bridge between systems; it is also an invitation to build a shared surface where multiple communities can meet.

For these reasons, execution is simultaneously a technical, social, and cultural project. The technical core can be specified and tested, but the ecosystem only becomes real when it is used: when people publish nodes, share runnable agents, teach through playgrounds, and improve the shared artefacts through criticism and implementation experience.

### 12.2.4 Closure

This chapter began by returning to a familiar diagram, not in order to introduce new ideas, but to reassess what has already been said from a different vantage point. The Prolog Trinity ecosystem, viewed in this way, is no longer merely a conceptual synthesis, but a set of commitments: to certain abstractions, to certain boundaries, and to a particular way of embedding logic and agency into the Web.

Execution, as it has been used throughout this chapter, should therefore not be understood as a final step that follows design. It is an ongoing process in which designs are tested, revised, and sometimes abandoned as they encounter real users, real systems, and real constraints. Some parts of the Trinity may mature quickly; others may remain speculative for a long time; still others may turn out to have been mistaken. That is not a failure of the vision, but an inevitable feature of any attempt to build shared infrastructure.

In the end, an ecosystem is not something one invents – it is something one manages to invite into existence. The chapter opened with banners because they are retrospective artefacts: they do not make a technology real, they mark that it has become shared. They compress years of discussion, specification work, test-suite building, implementation experience, and the slow accumulation of trust. In that sense, the banner motif was never a claim – it was a reminder of the kind of work that must happen outside the pages of this book if the Trinity is to become more than a proposal.

This returns us, quietly but insistently, to the warning implicit in the epigraph. Vision without execution is not a harmless state of incompleteness; it is a state in which ideas cannot be stress-tested, interoperable boundaries cannot be drawn, and responsibility cannot be shared. Execution is therefore not the last step after design, but the process through which design becomes accountable: to users, to implementors, to adversarial conditions, and to the constraints of the Web.

What ultimately matters, then, is not whether the Prolog Trinity ecosystem comes to resemble the diagram with which this chapter began, but whether it helps carve out a stable place for executable logic and communicative agents in the Web's future. A diagram can summarise an ambition, but only practice can decide what deserves to endure. If the Trinity proves to be a useful way of thinking – a way that enables real systems, attracts real collaborators, and clarifies real trade-offs – then execution will not be a moment of completion, but a gradual shift in what feels natural to build.

The comes the hard part: to get people to use the thing. For this, there are no guarantees. It may well turn out as a joke of a network. But at least we tried, and that is worth something.

# Part VII
# SCRAP

# Part VIII
# Appendices

# Appendix A
# Manual

## A.1 Predicates for programming with core actors

**Predicate:** `self/1` ACTOR

```
self(-Pid) is det.
```

Binds `Pid` to the process identifier of the calling process.

**Note:** The runtime uses a global pid model. Conceptually, pids are of the form `Id@Node`, although some compatibility-oriented surfaces, especially certain JSON responses, may still expose local pids as plain integers.

**Predicate:** `spawn/1-3` ACTOR

```
spawn(+Goal) is det.
spawn(+Goal, -Pid) is det.
spawn(+Goal, -Pid, +Options) is det.
```

Creates a new Web Prolog actor process running `Goal`. Valid options are:

- `node(+URI)`
  Creates the process locally or on a remote compatible node. Default is `localhost`.
- `monitor(+Boolean)`
  If `true`, monitoring is installed as part of process creation. Default is `false`.
- `link(+Boolean)`
  If `true`, installs directional parent-to-child link cleanup. Default is `true`.
- `load_text(+AtomOrString)`
  Loads the clauses specified by a Web Prolog source text into the actor's private Prolog database before calling `Goal`.
- `load_list(+ListOfClauses)`
  Converts the clauses to source text and loads them into the actor's private Prolog database before calling `Goal`.

- `load_uri(+URI)`
  Loads the clauses specified by a file path, file URI, or HTTP(S) URI into the actor's private Prolog database before calling `Goal`.
- `load_predicates(+ListOfPredicateIndicators)`
  Loads the local predicates denoted by `ListOfPredicateIndicators` into the actor's private Prolog database before calling `Goal`.

**Note:** In `spawn(+Goal, -Pid, +Options)`, the option `monitor(true)` installs monitoring as part of process creation. When the child terminates, the parent receives a message `down(Ref, Pid, Reason)` where `Pid` is the terminated actor, `Reason` is its exit reason, and `Ref` identifies the monitor instance. For monitoring created via `monitor(true)`, the current implementation uses `Ref = Pid`.

**Note:** The semantics of `link(true)` are directional: if a parent spawns a child with linking enabled, termination of the parent causes linked children to be terminated. The link does not imply symmetric bidirectional exit propagation.

**Note:** It is possible to pass an arbitrary number of the `load_*` options to `spawn/3`, and possibly more than one instance of each variant. To ensure that clauses end up in a well-defined order, they are converted into Prolog source text before being loaded into the database. The order of clauses and directives in the resulting source text is determined by the order of the `load_*` options in the option list.

**Predicate:** `monitor/2` ACTOR

```
monitor(+PidOrName, -Ref) is det.
```

Installs a monitor and returns a fresh reference in `Ref`. The first argument may be a pid or a registered local name. When the monitored process terminates, the monitoring process receives a message of the form `down(Ref, Pid, Reason)`.

**Predicate:** `demonitor/1-2` ACTOR

```
demonitor(+Ref) is det.
demonitor(+Ref, +Options) is det.
```

Stops monitoring identified by `Ref`. This is idempotent. There is only one valid option:

- `flush`
  Non-blocking removal of one pending `down(Ref, _, _)` message from the mailbox.

Using `monitor(true)` in `spawn/3` is the safest variant for short-lived children, since monitoring is established during spawn. Installing monitoring later with `monitor/2` is a separate step and may miss a child that exits immediately.

**Predicate:** `register/2` ACTOR

```
register(+Name, +Pid) is det.
```

Register a local process under a name, where `Name` is an atom and `Pid` identifies the actor process. The association between the name and the pid is removed when the process terminates.

**Predicate:** `whereis/2` ACTOR

```
whereis(+Name, ?Pid) is det.
```

Determine the identity of the actor process associated with the name. Binds `Pid` to `undefined` if the process does not exist.

**Predicate:** `unregister/1` ACTOR

```
unregister(+Name) is det.
```

Remove the association between the name and the process.

**Predicate:** `actors/1` ACTOR

```
actors(-Pids) is det.
```

Binds `Pids` to the list of actor pids visible from the current execution context. In ordinary public client execution, this means the current actor or toplevel together with any actors that have been spawned on behalf of the same client interaction. Actors belonging to other clients are not included.

**Note:** Outside public client execution, for example in node-owned runtime code or administrative code, `actors/1` returns the pids of all active local actors on the node.

**Predicate:** `exit/1` ACTOR

```
exit(+Reason) is det.
```

Exit the calling process with `Reason`.

**Predicate:** `exit/2` ACTOR

```
exit(+Pid, +Reason) is det.
```

Exit the process identified by `Pid` with `Reason`. For remote actors, the runtime routes this request through the remote node.

**Predicate:** `!/2, send/2-3` ACTOR

```
+PidOrName ! +Message is det.
send(+PidOrName, +Message) is det.
send(+PidOrName, +Message, +Options) is det.
```

Sends `Message` to the mailbox of the process identified as `PidOrName`. `PidOrName` may be a local pid, a local registered name, or a global pid of the form `Id@Node`. Valid options for `send/3` are:

- `delay(+Number)`
  Delays the sending by a specified number of seconds. Default is `0`.
- `id(+ID)`
  `ID` is a user-supplied identifier that can be used by `cancel/1` to stop the sending from taking place.

**Predicate:** `cancel/1` ACTOR

```
cancel(+ID) is det.
```

Tries to cancel the sending of *all* delayed messages with the specified ID. This is best-effort only, since a message may already have been sent by the time the call is made.

**Predicate:** `output/1-2` ACTOR

```
output(+Data) is det.
output(+Data, +Options) is det.
```

Sends a message `output(Pid,Data)` to the target process. `Pid` is the pid of the current process. Valid option:

- `target(+Pid)`
  Send the message to `Pid`. Default is the parent process.

Note that this is just a convenience predicate. A toplevel, like any other actor, may use `!/2` to send any term to any process to which it has a pid.

**Predicate:** `input/2-3` ACTOR

```
input(+Prompt, -Data) is det.
input(+Prompt, -Data, +Options) is det.
```

Sends a message `prompt(Pid,Prompt)` to the target process and waits for its input. `Prompt` may be any non-variable term. `Pid` is the pid of the current process. `Data` will be bound to the term that the target process sends using `respond/2`. Valid option:

- `target(+Pid)`
  Send the `prompt` message to `Pid`. Default is the parent process.

**Predicate:** `respond/2` ACTOR

```
respond(+Pid, +Input) is det.
```

Sends a response in the form of the term `Input` to a process that has prompted its parent process for input.

**Predicate:** `receive/1-2` ACTOR

```
receive(+Clauses) is semidet.
receive(+Clauses, +Options) is semidet.
```

`Clauses` is a sequence of receive clauses delimited by a semicolon:

```
{  Pattern1 [if Guard1] ->
         Body1 ;
   ...
   PatternN [if GuardN] ->
         BodyN
}
```

Each pattern in turn is matched against the first message in the mailbox. If a pattern matches and the corresponding guard succeeds, the matching message is removed from the mailbox and the body of the receive clause is called. If no message is accepted, the process waits for new messages, checking them one at a time in arrival order. Messages that are not accepted are *deferred*, i.e. left in the mailbox without any change in their contents or order. Valid options:

- `timeout(+Number)`
  If nothing appears in the current mailbox within `Number` seconds, the predicate succeeds anyway. Default is no timeout.
- `on_timeout(+Goal)`
  If the timeout occurs, `Goal` is called.

## A.2 Predicates for programming with toplevel actors

**Predicate:** `toplevel_spawn/1-2` ACTOR

```
toplevel_spawn(-Pid) is det.
toplevel_spawn(-Pid, +Options) is det.
```

Spawns a toplevel actor and binds `Pid` to its pid. All options accepted by `spawn/3` are also accepted by `toplevel_spawn/2`. In addition, `toplevel_spawn/2` accepts the following options:

- `session(+Boolean)`
  If set to `false`, the toplevel actor terminates after having run a goal to completion. If `true`, further interaction is expected. Default is `false` in the actor API.

- `target(+Pid)`
  Send all answer terms to `Pid`. Default is the pid of the parent.

**Predicate:** `toplevel_call/2-3` ACTOR

```
toplevel_call(+Pid, +Goal) is det.
toplevel_call(+Pid, +Goal, +Options) is det.
```

Asks the toplevel `Pid` for solutions to `Goal`. Valid options are:

- `template(+Template)`
  `Template` is a term sharing variables with `Goal`. By default, the template is identical to the goal.
- `offset(+Integer)`
  Collect only the slice of solutions starting from `Integer`. Default is `0`.
- `limit(+Integer)`
  Restrict the length of the returned list of solutions to `Integer`.
- `once(+Boolean)`
  When `true`, the list of solutions in a `success` answer term may not be the only solutions that `Goal` has, but the answer term may still indicate that it is. Default is `false`. Makes sense only in combination with the `limit` option.
- `target(+Pid)`
  Send the answer term to `Pid`. Default is the value of `target` when passed as an option to `toplevel_spawn/2`.

Variables in `Goal` are not bound directly in the caller. Instead, solutions and other kinds of output are returned in the form of answer messages delivered to the mailbox of the target process.

- `success(Pid, Data, More)`
  `Pid` is the pid of the toplevel process that succeeded in finding solutions to `Goal`. `Data` is a list holding instantiations of `Template`. `More` is either `true` or `false`, indicating whether the toplevel can return more solutions if `toplevel_next/1-2` is called.
- `failure(Pid)`
  `Pid` is the pid of the toplevel process that failed for lack of solutions.
- `error(Pid, Data)`
  `Pid` is the pid of the toplevel throwing the error. `Data` is the error term.

**Predicate:** `toplevel_next/1-2` ACTOR

```
toplevel_next(+Pid) is det.
toplevel_next(+Pid, +Options) is det.
```

Asks toplevel `Pid` for more solutions. Valid options:

- `limit(+Integer)`
  If omitted, the actor API keeps using the most recent limit value from the previous toplevel call state.

- `target(+Pid)`
  Send the answer term to `Pid`. Default is the value of `target` from the previous call state.

The messages delivered to the target mailbox are the same as for `toplevel_call/2-3`.

**Predicate:** `toplevel_stop/1` ACTOR

```
toplevel_stop(+Pid) is det.
```

Asks toplevel `Pid` to stop searching for more solutions.

**Predicate:** `toplevel_abort/1` ACTOR

```
toplevel_abort(+Pid) is det.
```

Tells toplevel `Pid` to abort the execution of the currently running goal.

## A.3 Predicates for programming with server actors

**Predicate:** `server_spawn/3-4` ACTOR

```
server_spawn(+Pred, +State, -Pid) is det.
server_spawn(+Pred, +State, -Pid, +Options) is det.
```

Spawns a generic server actor using callback predicate `Pred/4` and initial state `State`. The callback receives `(Request, OldState, Response, NewState)`. All options are forwarded to `spawn/3` except:

- `name(+Name)`
  Register the server under `Name`.

**Predicate:** `server_request/3-4` ACTOR

```
server_request(+To, +Request, -Response) is det.
server_request(+To, +Request, -Response, +Options) is det.
```

Makes a synchronous request-response call to the server `To`. `server_request/4` accepts the options of `receive/2`, for example `timeout(+Seconds)`. Monitoring is installed automatically so the call fails fast if the server terminates before replying.

**Predicate:** `server_promise/3-4` ACTOR

```
server_promise(+To, +Request, -Ref) is det.
server_promise(+To, +Request, -Ref, -MonRef) is det.
```

Sends `Request` to the server `To` and returns a correlation reference `Ref`. The reply must later be collected with `server_yield/2-4`. The four-argument variant additionally installs a monitor and returns its reference in `MonRef`.

**Predicate:** `server_yield/2-4` ACTOR

```
server_yield(+Ref, -Response) is det.
server_yield(+Ref, -Response, +Options) is det.
server_yield(+Ref, +MonRef, -Response, +Options) is det.
```

Waits for the response matching `Ref`. The three-argument variant accepts the options of `receive/2`. The four-argument variant uses `MonRef` to detect server termination and throws `server_down(Reason)` if the server dies before replying.

**Predicate:** `server_upgrade/2` ACTOR

```
server_upgrade(+To, +Pred) is det.
```

Replaces the callback predicate of the running server `To` without disturbing its current state. `Pred` must be arity 4.

**Predicate:** `server_stop/2` ACTOR

```
server_stop(+To, -Reply) is det.
```

Asks the server `To` to stop gracefully and binds `Reply` to its acknowledgement.

## A.4 Predicates for programming with statechart actors

**Predicate:** `statechart_spawn/1-2` ACTOR

```
statechart_spawn(-Pid) is det.
statechart_spawn(-Pid, +Options) is det.
```

Spawns a statechart actor and binds `Pid` to its pid.

`statechart_spawn/2` requires exactly one source option:

- `load_uri(+URI)`
  Load and run a statechart from a URI or file path.
- `load_text(+Text)`
  Load and run a statechart from an XML text string.

All remaining options are passed through to `spawn/3`.

`load_list/1` and `load_predicates/1` are rejected for `statechart_spawn/2`.

**Predicate:** `raise/1` STATECHART BUILTIN

```
raise(+Event) is det.
```

Enqueues `Event` on the *internal* event queue of the current statechart interpreter.

**Important:** `raise/1` is only meaningful inside executable statechart content such as <datamodel>, <onentry>, <onexit>, and <go>.

## A.5 Predicates for programming with supervisor actors

**Predicate:** `supervisor_spawn/2-3` ACTOR

```
supervisor_spawn(+ChildSpecs, -Pid) is det.
supervisor_spawn(+ChildSpecs, -Pid, +Options) is det.
```

Spawns a supervisor actor and starts its children. Supported options are:

- `strategy(+Strategy)`
  One of `one_for_one` (default), `one_for_all`, or `rest_for_one`.
- `intensity(+Integer)`
  Max restarts in period. Default `1`.
- `period(+Integer)`
  Window size for restart intensity. Default `5`.
- `name(+Name)`
  Register the supervisor under `Name`.

Other options are passed through to `spawn/3`.

Child specifications take the form `child(Id, ChildOptions)`. Child options are:

- `start(+Goal)`
  Required. Start goal for the child.
- `restart(+Policy)`
  One of `permanent` (default), `transient`, or `temporary`.
- `shutdown(+Shutdown)`
  One of `brutal_kill`, `infinity`, or a timeout number.
- `type(+Type)`
  `worker` (default) or `supervisor`.

**Predicate:** `supervisor_spawn_child/3` ACTOR

```
supervisor_spawn_child(+Pid, +ChildSpec, -Reply) is det.
```

Dynamically adds and starts one child. `Reply` is one of `ok`, `error(already_present)`, or `error(start_failed)`.

**Predicate:** `supervisor_terminate_child/3` ACTOR

```
supervisor_terminate_child(+Pid, +Id, -Reply) is det.
```

Stops child `Id` but keeps its specification. `Reply` is `ok` or `error(not_found)`.

**Predicate:** `supervisor_delete_child/3` ACTOR

```
supervisor_delete_child(+Pid, +Id, -Reply) is det.
```

Removes a non-running child specification. `Reply` is `ok`, `error(running)`, or `error(not_found)`.

**Predicate:** `supervisor_respawn_child/3` ACTOR

```
supervisor_respawn_child(+Pid, +Id, -Reply) is det.
```

Restarts a previously terminated child from its stored specification. `Reply` is either `ok(NewPid)`, `error(running)`, `error(not_found)`, or `error(start_failed)`.

**Predicate:** `supervisor_which_children/2` ACTOR

```
supervisor_which_children(+Pid, -Children) is det.
```

Returns a list of:

```
info(Id, Pid, Type, Restart)
```

where `Pid` may be `undefined` for stopped children.

**Predicate:** `supervisor_count_children/2` ACTOR

```
supervisor_count_children(+Pid, -Counts) is det.
```

Returns:

```
[specs-N, active-N, supervisors-N, workers-N]
```

**Predicate:** `supervisor_stop/1` ACTOR

```
supervisor_stop(+Pid) is det.
```

Stops the supervisor and shuts down children in reverse start order.

**Note on synchronous supervisor calls**

The dynamic/query API above uses monitored synchronous calls and may throw:

- `supervisor_down(Reason)`
- `supervisor_call_timeout(Sup, Request)`

## A.6 Predicates for remote Prolog-style calls

**Predicate:** `rpc/2-3` ISOBASE

```
rpc(+URI, +Goal) is nondet.
rpc(+URI, +Goal, +Options) is nondet.
```

Semantically, `rpc/2-3` executes a copy of `Goal` on the remote node identified by `URI` and then unifies the local goal with the remote copy on backtracking. The current implementation uses the stateless `/call` endpoint. The following options are valid:

- `limit(+Integer)`
  Restricts the number of solutions retrieved per roundtrip.
- `once(+Boolean)`
  When `true`, the returned slice may not be the only solutions that `Goal` has, but the answer term may still indicate that it is.
- `timeout(+Number)`
  Requests a timeout in seconds for goal execution on the remote node. The effective timeout is the minimum of the requested value and the timeout configured by the node owner.
- `http_timeout(+Number)`
  Sets the client-side HTTP transport timeout in seconds for each request.
- `load_text(+AtomOrString)`
  Loads source text into the underlying actor's private database before calling `Goal`.
- `load_list(+ListOfClauses)`
  Loads a list of clauses into the underlying actor's private database before calling `Goal`.
- `load_uri(+URI)`
  Loads source text from `URI` into the underlying actor's private database before calling `Goal`.
- `load_predicates(+ListOfPredicateIndicators)`
  Loads the local predicates denoted by the supplied predicate indicators into the underlying actor's private database before calling `Goal`.

`http_timeout/1` controls only the client-side HTTP transport timeout. It does not change the remote execution timeout.

**Predicate:** `promise/3-4` ISOBASE

```
promise(+URI, +Goal, -Ref) is det.
promise(+URI, +Goal, -Ref, +Options) is det.
```

Makes an asynchronous RPC call to node `URI` with `Goal`. The current implementation supports the following options:

- `template(+Template)`
  `Template` is a term sharing variables with the goal. By default, the template is identical to the goal.
- `offset(+Integer)`
  Collect only the slice of solutions starting from `Integer`. Default is `0`.
- `limit(+Integer)`
  Restrict the number of returned solutions to `Integer`.

The reference returned in `Ref` can later be used by `yield/2-3` to collect the answer.

**Predicate:** `yield/2-3` ISOBASE

```
yield(+Ref, ?Answer) is det.
yield(+Ref, ?Answer, +Options) is det.
```

Returns the promised answer from a previous call to `promise/3-4`. This predicate must be called by the same process from which the previous call to `promise/3-4` was made. Valid options:

- `timeout(+Number)`
  If nothing appears in the current mailbox within `Number` seconds, the goal supplied in the `on_timeout` option is called. Default is no timeout.
- `on_timeout(+Goal)`
  If the timeout occurs, `Goal` is called.

## A.7 Built-in predicates for node deployment

**Predicate:** `node/1-2` ISOBASE

```
node(+Port) is det.
node(+Port, +Options) is det.
```

Starts an HTTP server for a Prolog node on `Port`.

In the current implementation, the node exposes:

- `/`
- `/call`
- `/toplevel_spawn`
- `/toplevel_call`
- `/toplevel_next`

- `/toplevel_poll`
- `/toplevel_stop`
- `/toplevel_abort`
- `/toplevel_respond`
- `/shell`
- `/ws`

The node module defines two owner-controlled settings:

- `node:cache_size(+Integer)`
  Maximum number of cache entries used by the stateless `/call` endpoint. Default is `100`.
- `node:timeout(+Number)`
  Owner timeout cap in seconds. Default is `2`. If a request includes a timeout value, the effective timeout is the minimum of that requested value and `node:timeout`.

`node/2` accepts shared-database startup options:

- `load_shared_db_text(+Text)`
- `load_shared_db_file(+File)`
- `load_shared_db_uri(+URI)`

The shared database is loaded once into a dedicated runtime module and imported by actor modules.

## A.8 The stateful WebSocket API

The WebSocket ACTOR profile lives at `/ws`. Messages are encoded as JSON dicts.

Supported commands from client to server are:

- `toplevel_spawn`
- `toplevel_call`
- `toplevel_next`
- `toplevel_stop`
- `toplevel_abort`
- `toplevel_respond`
- `spawn`
- `send`
- `exit`

Supported event types from server to client are:

- `spawned`
- `success`
- `failure`

- `error`
- `output`
- `prompt`
- `timeout`
- `stop`
- `abort`
- `responded`
- `down`

In WebSocket payloads, pid values may appear as integers for local actors or as strings of the form `"Id@Node"` for non-local actors.

## A.9 The semi-stateful HTTP API

### I/O and long-polling

The semi-stateful API is built on long-lived toplevel actors plus per-session queues.

Table A.1 lists the parameters for the `/toplevel_spawn` endpoint.

| Parameter | Type | Description | Default Value |
|---|---|---|---|
| `format` | atom | Response format, typically `json` or `prolog`. | `json` |
| `options` | list or string | Spawn options passed to the underlying toplevel actor. | `[]` |
| `load_text` | string | Source text to be loaded into the session actor before use. | empty string |

**Table A.1** HTTP API parameters for the `/toplevel_spawn` endpoint.

**Note:** In the actual HTTP ISOTOPE API, `session(true)` is always forced by the endpoint, regardless of any `session/1` value that may be present in `options`. The endpoint also injects a small prelude that redirects `writeln/1` to `actor:output/1`.

`/toplevel_poll` takes `pid`, optional `format`, and optional `timeout`. `/toplevel_respond` takes `pid`, `input`, and optional `format`. `/toplevel_stop` and `/toplevel_abort` take `pid` and optional `format`.

Table A.4 lists the complete set of endpoints and their current responses in the semi-stateful API.

| Parameter | Type | Description | Default Value |
|---|---|---|---|
| `pid` | pid | The toplevel actor to engage. Accepts integer forms and canonical `Id@Node` forms. | None |
| `goal` | callable | The goal to be called. | None |
| `template` | term | The template to be used. For `format=json`, named query variables are returned as a JSON dict template. | Same as goal |
| `offset` | int >= 0 | The offset in the virtual list of solutions. | 0 |
| `limit` | int > 0 | Max number of solutions to be returned. | Effectively no limit |
| `format` | atom | Response format. | `json` |
| `load_text` | string | Additional session-local source text to be loaded before the call. | empty string |
| `once` | boolean | Restrict execution to one slice even if more solutions may exist. | `false` |
| `timeout` | number | Requested wait timeout for this call, capped by the node owner timeout. | Owner timeout |

**Table A.2** HTTP API parameters for the `/toplevel_call` endpoint.

| Parameter | Type | Description | Default Value |
|---|---|---|---|
| `pid` | pid | The toplevel actor to continue. Accepts integer forms and canonical `Id@Node` forms. | None |
| `limit` | int > 0 | Max number of solutions to be returned by this HTTP wrapper call. | Effectively no limit |
| `format` | atom | Response format. | `json` |
| `timeout` | number | Requested wait timeout, capped by the node owner timeout. | Owner timeout |

**Table A.3** HTTP API parameters for the `/toplevel_next` endpoint.

## A.10 The stateless HTTP API

The stateless API is exposed at `/call`.
The only required parameter is `goal`.

Although the API is stateless from the client's perspective, the server may cache a live toplevel actor internally to continue paged solution retrieval without recomputing earlier work.

| API endpoint | Response types |
|---|---|
| /toplevel_spawn | spawned or error |
| /toplevel_call | success, failure, error, output, prompt, timeout, or abort |
| /toplevel_next | success, failure, error, output, prompt, timeout, or abort |
| /toplevel_respond | responded or error |
| /toplevel_poll | success, failure, error, output, prompt, timeout, or abort |
| /toplevel_stop | stop |
| /toplevel_abort | abort |

**Table A.4** The complete set of endpoints and responses in the semi-stateful API.

| Parameter | Type | Description | Default Value |
|---|---|---|---|
| goal | callable | The goal to be called. | None |
| template | term | The template to be used. For format=json, named query variables are returned as a JSON dict template. | Same as goal |
| offset | int >= 0 | The offset in the virtual list of solutions. | 0 |
| limit | int > 0 | Max number of solutions to be returned. | Effectively no limit |
| load_text | string | Source text to be injected into the temporary actor context. | empty string |
| timeout | number | Requested timeout for remote goal execution, capped by the node owner's timeout setting. | Owner timeout setting |
| once | boolean-ish atom | Restrict execution to one slice even if more solutions may exist. | false |
| format | atom | The format of answer responses. | json |

**Table A.5** HTTP API parameters for the /call endpoint.

# Appendix B
# Glossary

This appendix collects working definitions for core terms used across the book and the current node implementation. The bias is conceptual first and operational second: each entry states what the term means in the book and, when useful, how the present implementation realizes it.

**Actor.** An actor is a long-lived concurrent process with a pid, a mailbox, and an explicit message interface. In the ACTOR profile, actors are the basic unit of concurrency. Toplevels, statechart processes, and node-resident services are all specialized actor forms.

**Agent.** An agent is a persistent interactive process that participates in ongoing interaction with its environment. In this book, a Prolog agent is defined by its interface and behavior rather than by its implementation language: it talks Prolog terms to other agents by receiving goals or messages and returning answers, messages, or effects in the same term-based form.

Actors, toplevels, and nodes can all count as agents, but they play different roles. Actors are loci of concurrent computation, toplevels expose a conversational interface, and nodes host knowledge, expose web APIs, and enforce policy.

**Capability.** A capability is authority to do or affect something. In the book, the term is often used in the capability-security sense: possession of an unforgeable or hard-to-guess handle, such as a pid or a reply channel, confers some authority. In the current node implementation, the auth layer also uses named policy capabilities such as `execute`, `admin`, and `internal_transport`, attached to a principal. These are different mechanisms, but they answer the same question: what authority is being granted?

**Containment.** Containment is the structural security discipline that keeps client-created computation bounded in lifetime and effect. A contained computation may still do real work, but it should not outlive its session, escape its authority bounds, or leave behind uncontrolled background activity.

In the current node, containment is reflected in the default rule that client-spawned actors are linked to the client or session that created them. Long-lived detached services are owner-managed exceptions, not the default.

**Federation.** Federation is the decision to standardize what systems share at the boundary rather than to force them to become internally identical. In the Trinity, this means shared web-facing interaction surfaces and execution profiles, while leaving room for different Prolog systems to diverge internally.

**Instant portability.** Instant portability is the limiting case of portability in which relocation requires no adaptation step. A program can be downloaded from one node and run locally, or shipped to another node, immediately because everything it depends on is either bundled with it or guaranteed by the target profile.

**Interaction surface.** An interaction surface is the externally visible contract through which systems meet: routes, message formats, APIs, protocol rules, and profile promises. The book uses this term to emphasize that interoperability depends on shared boundaries, not on shared internals.

**Interoperability.** Interoperability is the ability of independently implemented systems to work together correctly through shared protocols, representations, and profile contracts. In this project, it is a stronger notion than mere syntactic compatibility: two nodes interoperate when a client can rely on the same observable behavior across them at the advertised profile boundary.

**Inexpressibility.** Inexpressibility is the structural security property that certain dangerous operations cannot be expressed at all in client code. Rather than relying only on runtime refusal, the exposed language fragment excludes operations such as direct file I/O, raw operating-system access, or other ambient-authority mechanisms that would be unsafe to hand to arbitrary clients.

In the book, this is the sandbox side of the security story. In the current node, profile checks and sandbox policy are the implementation mechanisms that approximate this boundary.

**Invisibility.** Invisibility is the structural security property that a client can interact only with actors or services whose handles or names have been intentionally revealed. It is the name-discipline side of security: what cannot be named cannot be addressed.

In the book, opaque pids and owner-curated service publication are the key mechanisms. In the current node, this also means that published services are not the same thing as the ordinary client name registry.

**Node.** A node is a network-addressable host of execution, knowledge, and policy. A node exposes Web Prolog APIs, advertises a profile, enforces authentication and authorization policy, and may host both shared predicates and node-resident actors.

**Owner.** An owner is the authority responsible for configuring a node's public surface and long-lived resources. The owner role includes installing shared predicates, publishing or withdrawing services, changing node settings and policy, and taking responsibility for durable service lifecycle concerns such as supervision and recovery.

In the current implementation, owner authority is realized through administrative policy rather than through ordinary client capabilities. It is best understood as a role in the security model, not merely as "the client who happens to connect first."

**Pid.** A pid is an opaque process identifier for an actor. It is a routable handle, not a descriptive name, and in security terms it often functions as a capability. Clients should treat a pid as an implementation handle that may be passed around or used for communication, but not as something that should be globally enumerable.

**Private database.** A private database is the actor-local body of clauses and facts associated with an actor. It forms part of that process's private workspace rather than the node's shared knowledge, and it is not directly accessible to other actors.

In the book, the private database is treated as a capability rather than as a defining feature of agency: some agents use it heavily, while others remain effectively stateless. In practical terms, it can be populated at spawn time by `load_*` options, it may shadow definitions from the shared database, and updates to it affect only that actor.

**Portability.** Portability is the extent to which code, programs, or interaction patterns can move between nodes or implementations without semantic breakage or manual rewriting. In the book, portability is pursued by standardizing profiles and their mandatory capabilities rather than by demanding total internal uniformity across Prolog systems.

**Principal.** A principal is the identity on whose behalf a request is made. It is not the same thing as a user, session, connection, or pid. A principal may be an anonymous caller, a human user, a service account, or a trusted internal transport identity.

In the current node implementation, principals are resolved by the auth layer and mapped to policy capabilities. Authorization answers what a principal may do; it does not by itself determine which pid or session is involved.

**Profile.** A profile is a deliberately scoped contract describing which language features, APIs, and interaction patterns are available. Profiles let Web Prolog specify a shared operational subset without turning Web Prolog into a separate fork of Prolog.

In the current node implementation, the configured node profile is node-scoped, not principal-scoped. It governs which routes and goal forms are in profile, and public execution checks enforce that boundary.

**Public surface.** A node's public surface is the set of predicates, services, routes, and policies that it intentionally exposes to clients. It is smaller than the total internal runtime and should be treated as an explicit design choice rather than as an accident of what happens to be running.

**Registered name.** A registered name is a symbolic name bound to an actor. Ordinary registered names are convenience names for actor lookup and messaging; they are not necessarily public services.

Under the current regime, ordinary client registration and published service registration are separate. Client-side `register/2`, `whereis/2`, and `unregister/1` operate within a client-visible naming scope, while published services use an owner-controlled service registry.

**Reply channel.** A reply channel is a communication endpoint created for the purpose of receiving one reply or a narrowly bounded class of replies. In capability terms, it should carry reply authority only, not general authority to invoke the client.

The book treats reply channels as a way to avoid leaking a client's full pid to an untrusted peer. The general principle is that communication authority should be as narrow and short-lived as the protocol step that requires it.

**Service.** A service is a durable, shared conversational resource intentionally offered by a node for repeated use by multiple clients. In the Prolog Web, services may be predicate-based or actor-based, but node-resident actor services are especially important because they provide shared state or shared coordination over time.

In the current implementation, a published actor service is not just any actor with a convenient name. It is something the node owner explicitly publishes under a stable service name.

**Server.** A server, in the sense used by the book's generic-behavior chapters, is a stateful actor that maintains private state and serves clients through a stable request-response message protocol. The generic server behavior factors out the machinery of message routing, state threading, and reply handling so that the application-specific logic can be supplied as a callback predicate.

This is an Erlang-style behavior: it exposes a protocol, not a predicate-level logical interface, and it does not bind the caller's variables directly.

**Service discovery.** Service discovery is the process by which clients learn which services a node intentionally makes available. In the Prolog Web, discovery should be treated as an owner-curated publication mechanism, not as process introspection.

The important distinction is that discovery resolves published service names without revealing everything that happens to be running. In the current book example, this is realized declaratively through relations such as `service/2` queryable via `rpc/2-3`.

**Service publication.** Service publication is the act by which a node owner places a name into the node's public service namespace. Publication is stronger than ordinary naming: it makes a service part of the node's advertised public surface.

In the current regime, this is distinct from ordinary `register/2`. Client registration is for client-managed actors, while published services are managed through the owner-controlled service registry.

**Session.** A session is a bounded continuing interaction context in which state can persist across multiple protocol steps. The book contrasts session-oriented interaction with purely stateless request-response exchange.

In the current node, HTTP ISOTOPE sessions and WebSocket ACTOR sessions are both session forms, but they have different timing and failure boundaries.

**Shared database.** The shared database is the node-owned body of predicates and facts made available as shared knowledge. It is loaded once at node startup into a shared runtime module and then imported by actor modules or used by node-level query execution.

Clients may query against shared knowledge and combine it with private code, but the shared database is not the same thing as a client's private mutable workspace.

**Statechart actor.** A statechart actor is a Web Prolog actor whose control logic is given by a statechart document. It behaves like an ordinary actor with a mailbox, but incoming messages are interpreted as events that drive a structured state machine rather than an ad hoc `receive/1-2` loop.

The distinctive semantic idea is run-to-completion: processing one external event may trigger a finite cascade of internal work before the next external message is consumed. This makes control explicit and inspectable while keeping the process inside the ordinary actor model.

**Supervisor.** A supervisor is an actor whose main responsibility is to start, monitor, and manage child actors, especially in the face of failure. In the book's Erlang-style account, a supervisor embodies the "let it crash" discipline: workers are allowed to fail, and the supervisor applies the configured recovery policy.

Supervisors matter most for durable node-resident services, where restart and cleanup are part of the node owner's operational responsibility rather than something each worker should reimplement ad hoc.

**Toplevel actor.** A toplevel actor is an actor whose behavior is governed by an explicit conversational protocol for submitting goals, receiving answers incrementally, handling output, and managing abort or stop. It turns the ordinary Prolog toplevel into a first-class actor process.

In implementation terms, toplevel actors underpin `rpc/2-3`, the semi-stateful HTTP API, and much of the browser-facing shell machinery.

**PTCP.** PTCP is the Prolog Toplevel Communication Protocol. It is the explicit protocol governing conversation between a client and a toplevel actor: how goals are submitted, how answers are streamed incrementally, how output and prompts may occur, and how abort, stop, and exit are handled.

Its importance in the book is that it turns informal REPL behavior into a first-class behavioral contract. In implementation terms, PTCP underlies the observable behavior of toplevel actors across the stateless, semi-stateful, and stateful APIs.

# Appendix C
# How to implement a Prolog node

> Vision without execution is just hallucination.
>
> *Thomas Alva Edison*

We would have loved to be able to present a stable, speedy and secure implementation of a Prolog node, ready to be deployed to help building the Prolog Web. However, there exists no such implementation at this point in time. There are some proof-of-concept implementations, but they are neither stable nor speedy, nor secure. How can we build one that is? And how can we build *more* than one, so that we can make sure that interoperability across different implementations works as intended?

We also say something about browser-based implementations of Web Prolog and briefly discuss the prospects for developing a virtual machine with properties similar to Erlang's BEAM VM.[1]

## C.1 Wrapping a node around an existing Prolog system

> Make it work, then make it beautiful, then if you really, really have to, make it fast. 90% of the time, if you make it beautiful, it will already be fast. So really, just make it beautiful!
>
> *Joe Armstrong*

It is likely that the first implementations of Prolog nodes would be Prolog systems providing as libraries whatever is required to comply with Web Prolog requirements. This is how our proof-of-concept implementations were built.

Some really excellent Prolog systems exist out there, so if you are a Prolog implementer, one obvious advantage with this approach is that most of the necessary

[1] `https://en.wikipedia.org/wiki/BEAM_(Erlang_virtual_machine)`

work has already been done. The amount of additional work required to implement a node depends on which system it is built on top of.

In this section, we look at different ways to wrap a node around a system that supports the ISO Prolog working draft for threads.[2] We are aiming for an almost complete ISOBASE node, and an ACTOR node, albeit less complete.

Using SWI-Prolog, we show a way to implement the stateless HTTP API. We do not focus solely on semantics, but on performance too. In particular, we devise a way to optimize the API by avoiding the unnecessary recomputation of solutions that a naive implementation would have to do. Furthermore, we implement a version of `rpc/2-3` on top of the stateless HTTP API.

We implement specifications for how we believe predicates such as `spawn/1-3`, `!/2` and `receive/1-2` should work. On top of actors, we implement the behavior of Prolog toplevels. These implementations focus on semantics rather than performance.

We are seeking, if not beauty, then at least as much clarity and simplicity as possible. Our implementations are only partial, but we also indicate what else would be needed to complete them.

### C.1.1 Implementing an ISOBASE node

A Prolog ISOBASE node is equipped with a stateless HTTP API. Managing this API is actually the only task its node controller is resonsible for. It means that we can make good use of a library for building web servers. Here is how a web server may be written in SWI-Prolog using `library(http/http_server)`:

```
:- use_module(library(http/http_server)).

:- http_handler(root(call), node_controller_isobase, []).

node_controller_isobase(Request) :-
    http_parameters(Request, [
        goal(GoalAtom, [atom]),
        template(TemplateAtom, [default(GoalAtom)]),
        offset(Offset, [integer, default(0)]),
        limit(Limit, [integer, default(10 000 000 000)]),
        format(Format, [atom, default(json)])
    ]),
    atomic_list_concat([GoalAtom,+,TemplateAtom], QTAtom),
    read_term_from_atom(QTAtom, Goal+Template, []),
    compute_answer(Goal, Template, Offset, Limit, Answer),
    respond_with_answer(Format, Answer).
```

[2] `http://logtalk.org/plstd/threads.pdf`

```
node(Port) :-
    http_server(http_dispatch, [port(Port)]).
```

The call to `compute_answer/5` is responsible for the real work here. It takes a goal, a template, an offset and a limit, and computes an answer term serving as a response to the request which can be sent back to the client formatted as Prolog or JSON. There is more than one way to implement this predicate. Let us first look at a simple (but from a performance point of view naive) way of doing it.

SWI-Prolog offers a library predicate `findnsols/4` which provides a useful foundation for our implementation. It is somewhat similar to the standard `findall/3`, but expects an integer `Limit` in its first argument and will generate at most that many solutions. It is also nondeterministic, so on backtracking it will do it again. We borrow an example of its use from the SWI-Prolog manual:[3]

```
?- findnsols(5, I, between(1, 12, I), L).
L = [1, 2, 3, 4, 5] ;
L = [6, 7, 8, 9, 10] ;
L = [11, 12].
?-
```

Another SWI-Prolog library predicate `offset/2` will also prove useful.[4] Its purpose is to *skip* the first $n$ solutions to a goal, i.e. the first $n$ solutions are computed, but not collected. Here is an example of its use:

```
?- offset(10, between(1, 12, I)).
I = 11 ;
I = 12.
?-
```

Combining `findnsols/4` with `offset/2` allows us to implement a predicate `slice/5` capable of computing a *slice* of solutions to a goal:

```
slice(Goal, Template, Offset, Limit, Slice) :-
   findnsols(Limit, Template, offset(Offset, Goal), Slice).
```

However, we are looking for *answers*, rather than just slices of solutions. By wrapping a call to `slice/5` in a call to `call_cleanup/2` wrapped by a call to `catch/3` we arrive at a predicate `answer/5` capable of producing the four different forms of answer terms that we need:

```
answer(Goal, Template, Offset, Limit, Answer) :-
   catch(
      call_cleanup(slice(Goal, Template, Offset, Limit, Slice),
                   Det = true),
         Error, true),
```

[3] `https://www.swi-prolog.org/pldoc/doc_for?object=findnsols/4`

[4] https://www.swi-prolog.org/pldoc/doc_for?object=offset/2

```
   (   Slice == []
   ->  Answer = failure
   ;   nonvar(Error)
   ->  Answer = error(Error)
   ;   var(Det)
   ->  Answer = success(Slice, true)
   ;   Det = true
   ->  Answer = success(Slice, false)
   ).
```

This predicate will turn out to be useful in more than one way. In this context it will be used for the implementation of `compute_answer/5`. In this role we want `compute_answer/5` to be deterministic, so since the call to `answer/5` is non-deterministic we need to wrap it in a call to `once/1` like so:

```
compute_answer(Goal, Template, Offset, Limit, Answer) :-
    once(answer(Goal, Template, Offset, Limit, Answer)).
```

The implementation of our simple but naive stateless HTTP API is almost complete, and assuming we also have a suitable implementation of `respond_with_answer/2`, we can now start running a node:

```
?- node(3010).
% Started server at http://localhost:3010/
true.
?-
```

At this point we may want to take the node's stateless HTTP API for a trial run by entering the following URI in a web browser:

```
http://localhost:3010/call?goal=member(X,[a,b])&format=prolog
```

In the browser's window, we should then see the following:

```
success([member(a,[a,b]),member(b,[a,b])],false)
```

By appending `&template=X&offset=0&limit=1` to the URI we should get

```
success([a],true)
```

and by incrementing the `offset` parameter by 1 we should see

```
success([b],false)
```

Note that it is important that we do not expose the node to the whole world at this point, as it is not secure.

### C.1.2 Implementing rpc/2-3 on top of the stateless HTTP API

As soon as we have an implementation of the stateless HTTP API, we can easily, by means of two other libraries provided by SWI-Prolog,[5] implement `rpc/2-3` on top of it. Here is the source code:

```
:- use_module(library(http/http_open)).
:- use_module(library(url)).

rpc(URI, Goal) :-
    rpc(URI, Goal, []).

rpc(URI, Goal, Options) :-
    parse_url(URI, Ps),
    term_variables(Goal, Vars),
    Template =.. [v|Vars],
    format(atom(GA), "(~p)", [Goal]),
    format(atom(TA), "(~p)", [Template]),
    option(limit(L), Options, 10 000 000 000),
    rpc_7(Template, 0, L, GA, TA, Ps, Options).

rpc_7(Template, O, L, GA, TA, Ps, Os) :-
    parse_url(ExpandedURI, [
        path('/call'),
        search([goal=GA, template=TA, offset=O,
                limit=L, format=prolog])
      | Ps
    ]),
    setup_call_cleanup(
        http_open(ExpandedURI, Stream, Os),
        read(Stream, Answer),
        close(Stream)),
    rpc_8(Answer, Template, O, L, GA, TA, Ps, Os).

rpc_8(success(Slice, true), Template, O, L, GA, TA, Ps, Os) :-  !,
    (   member(Template, Slice)
    ;   NewO is O + L,
        rpc_7(Template, NewO, L, GA, TA, Ps, Os)
    ).
rpc_8(success(Slice, false), Template, _, _, _, _, _, _) :-
    member(Template, Slice).
rpc_8(failure, _, _, _, _, _, _, _) :- fail.
rpc_8(error(E), _, _, _, _, _, _, _) :- throw(E).
```

[5] See `https://www.swi-prolog.org/pldoc/man?section=httpopen` and `https://www.swi-prolog.org/pldoc/man?section=url`

The idea behind this code is to use `http_open/3` in a loop in order to make one or more requests for consecutive slices of solutions to the goal in the first argument using the stateless HTTP API. The URI of each request takes the form

```
BaseURI/call?goal=G&template=T&offset=O&limit=L&format=prolog
```

where `O` is initially `0` and is incremented by `L` between requests.

The most interesting parts of the implementation are the use of the disjunction in the body of the first `rpc/7` clause and the use of `member/2` in the first and second clauses. They are responsible for turning the responses to the deterministic requests made by `http_open/3` into the nondeterministic behavior we want `rpc/2-3` to show.

Let us test our implementation by running an example from Chapter 4, showing how `rpc/2-3` can be used:

```
?- [user].
|: human(plato).
|: human(aristotle).
|: ^D% user://1 compiled 0.00 sec, 2 clauses
true.
?- rpc('http://localhost:3010', human(Who)).
Who = plato ;
Who = aristotle.
?-
```

Note that although the query has two solutions, only one network roundtrip is made, triggered by the following HTTP request:

```
GET http://localhost:3010/call?goal=human(Who)&format=prolog
```

The response contains the following answer term:

```
success([human(plato),human(aristotle)],false)
```

The above code is just a sketch that leaves out some of the details that are necessary for a fully working node. In particular, it does not implement `respond_with_answer/2` and it does not handle syntax errors in queries. None of this would be difficult to add, and with such additions, this section together with the previous one implements the stateless API of an ISOBASE node, as well as the `rpc/2-3` predicate.

### C.1.3 Fixing a problem due to spurious recomputation

The above implementation of the HTTP API suffers from a performance problem. The problem is easy to spot when timing a goal simulating a situation where a first solutions takes a long time to compute while a second solution takes almost no time at all – a goal such as the disjunction `(sleep(1),X=foo;X=bar)` for example. Here is how this looks in a system such as SWI-Prolog:

```
?- time((sleep(1), X=foo ; X=bar)).
% 1 inferences, 0.000 CPU in 1.005 seconds
X = foo ;
% 7 inferences, 0.000 CPU in 0.000 seconds
X = bar.
?-
```

As expected, solving the first disjunct took one second, while the second disjunct took almost no time at all. However, when calling this goal using `rpc/3` with the `limit` options set to 1, we see the following:

```
?- _URI = 'http://localhost:3010',
   time(rpc(_URI, (sleep(1), X=foo ; X=bar), [limit(1)])).
% 1,984 inferences, 0.001 CPU in 1.006 seconds
X = foo ;
% 1,804 inferences, 0.001 CPU in 1.009 seconds
X = bar.
?-
```

The cause of this problem lies not in the implementation of `rpc/2-3`, but in the HTTP API, and more precisely in the way `compute_answer/5` works. Consider the following call, where the third argument (for the offset) is 1:

```
?- _Goal = (sleep(1), X=foo ; X=bar),
   time(compute_answer(_Goal, X, 1, 1, Answer)).
% 30 inferences, 0.000 CPU in 1.005 seconds
Answer = success([bar], false).
?-
```

In general, computing the first slice (i.e. the one starting at offset 0) is as fast as it can be, but computing the second slice involves the recomputation of the first slice and, more generally, computing the *nth* slice involves the recomputation of all preceding slices, the results of which are then just thrown away. This, of course, is a waste of resources and puts an unnecessary burden on the node.

This is not as bad as it looks. Most uses of `rpc/2-3` will compute all solutions at once and thus make only one network roundtrip.

```
?- time(rpc($_URI, (sleep(1), X=foo ; X=bar))).
% 2,011 inferences, 0.001 CPU in 1.007 seconds
X = foo ;
% 5 inferences, 0.000 CPU in 0.000 seconds
X = bar.
?-
```

As can be seen in the following example run, it is only when more than one network roundtrip may have to be made, so that the `limit` option must be employed, that the problem surfaces:

```
?- _Goal = (sleep(1), X=foo ; X=bar),
   time(compute_answer(_Goal, X, 0, 2, Answer)).
% 29 inferences, 0.000 CPU in 1.002 seconds
Answer = success([foo, bar], false).
?-
```

Still, to achieve a less wasteful and more efficient stateless querying even when more than one network roundtrip must be made, recomputation of the kind described in the previous section should be avoided. In this section we lay out an approach where the node controller (subject to a setting) may *cache* the state of the toplevel process that produced the *nth* slice of solutions to a query, so that the work spent on producing it will not have to be repeated. This can still be done without requiring that the node controller remembers *which* client made the request for the previous slices of solutions.

The method can be seen as a kind of *pooling* of toplevel processes, but while pooling usually involves a pool of merely initialized but idle processes which stand ready to be given work, this method involves a pool where each member has already done some real work. In other words, the idea here is not to cache *already computed* solutions but rather to cache the *potential* for new solutions in the form of processes that are idle, but have "more to give" if put to work.[6]

A consequence of this approach is that it allows the computation of the full set of solutions to a query to be distributed over more than one toplevel process. We can avoid spawning a new process for each incoming request, but instead, when available, select a member from a pool of suspended processes which, since it has already performed some of the work, needs to do as little as possible in order to compute the requested solutions. Using this approach, it is likely (although not guaranteed) that the work that generated the *nth* slice of solutions does not have to be repeated if a request for the next slice is made.

Check the principle of locality in https://chatgpt.com/share/68adf7f2-0b2c-24dc-800f-a795-94d2958c9671

One way to realize this is to make the node controller responsible for the maintenance of a cache consisting of entries pointing to members of the pool of suspended processes. Such a cache has a very straightforward implementation in Prolog thanks to its dynamic database. The signature of a cache entry can be given as follows:

```
cache(+Gid, +N, -Pid) is nondet.
```

Here, `Gid` is an identifier representing a goal `G` and a template `T`. `N` is an integer > `0`, and `Pid` is the pid of an already spawned process which, after having computed `N` solutions to `G` and returned them to the client, is now suspended but can be activated again at any point. A cache is simply a dynamic predicate comprising an ordered sequence of `cache/3` clauses. The cache will be searched from the top, stopping when the first match is found. Updates will be added to the bottom.[7]

The cache forms a queue-like data structure and can be seen as a kind of priority queue. When a request comes in which specifies a goal, a template, and an offset ¿ 1,

[6] Credits for this idea goes to Jan Wielemaker. The implementation is our's.

[7] Note that the implementation of the cache as a Prolog predicate is not mandated. A node would be free to implement it in a way that suits the host platform best.

the cache is scanned from the beginning of the queue, the first matching entry is dequeued, and the corresponding process is employed. If no matching entry is found, a new process is spawned. Newly created as well as updated cache entries are added to the end of the queue.

The maximum size of the cache for a particular node can be specified by its owner by means of a setting. What is a reasonable size depends on the host platform of the node, and in particular on the cost of keeping suspended toplevel actors around.

Here is an implementation of two predicates for managing the cache:

```
:- dynamic cache/3.

cache_retract(Gid, N, Pid) :-
    once(retract(cache(Gid, N, Pid))).

cache_update(Gid, N, Pid) :-
    assertz(cache(Gid, N, Pid)),
    setting(cache_size, Size),
    predicate_property(cache(_,_,_),
                       number_of_clauses(N)),
    N > Size -> cache_retract(_,_,_) ; true.
```

To ensure efficient cache lookup, the goal identifier `Gid` is a hash value computed from a grounded copy of the goal. In SWI-Prolog, `goal_id/2` may be implemented as follows:

```
goal_id(GoalTemplate, Gid) :-
    copy_term(GoalTemplate, Gid0),
    numbervars(Gid0, 0, _),
    term_hash(Gid0, Gid).
```

Equipped with the above utility predicates, `compute_answer/5` can be implemented like so:

```
compute_answer(Goal, Template, Offset, Limit, Answer) :-
    goal_id(Goal-Template, Gid),
    (   cache_retract(Gid, Offset, Pid)
    ->  thread_self(Self),
        toplevel_next(Pid, [
            limit(Limit),
            target(Self)
        ])
    ;   toplevel_spawn(Pid, [session(false)]),
        toplevel_call(Pid, Goal, [
            template(Template),
            offset(Offset),
            limit(Limit)
        ])
```

```
    ),
    setting(timeout, Timeout),
    receive({
        success(Pid, Slice, true) ->
            Index is Offset + Limit,
            cache_update(Gid, Index, Pid),
            Answer = success(Slice, true);
        success(Pid, Slice, false) ->
            Answer = success(Slice, false);
        failure(Pid) ->
            Answer = failure;
        error(Pid, Error) ->
            Answer = error(Error)
    },[
        timeout(Timeout),
        on_timeout((Answer = error(timeout),
                    exit(Pid, kill)))
    ]).
```

Given a goal and a template, a goal identifier `Gid` is computed. Since more than one client may request the same slice of solutions, the Gid is not unique. Based on the gid and the value of the `offset` parameter, an attempt to look up a cache entry pointing to a suitable toplevel process will be made. If this succeeds, `toplevel_next/2` will be called, which will compute an answer holding a slice of solutions no longer than the value of the `limit` parameter specifies. If it fails, a new toplevel will be spawned using `toplevel_spawn/3`, and `toplevel_call/3` will be called, which will compute the answer instead. *Note.* In this implementation, `toplevel_call/3` reads only the options `template/1`, `offset/1`, `limit/1`, `once/1`, and `target/1`. Other options in the list are silently ignored. This keeps wrappers simple, but misspelled option names will also be ignored.

The answer term resulting from this is sent to the thread in which the request handler is running and can be caught by `receive/2`. Note that if the reception of the term takes too long, it will result in a timeout error.

```
?- _Goal = (sleep(1), X=foo ; X=bar),
   time(compute_answer(_Goal, X, 0, 1, Answer)).
% 30 inferences, 0.000 CPU in 1.005 seconds
Answer = success([foo], true).
?-

...

?- _Goal = (sleep(1), X=foo ; X=bar),
   time(compute_answer(_Goal, X, 1, 1, Answer)).
% 30 inferences, 0.000 CPU in 0.005 seconds
Answer = success([bar], false).
?-
```

How can we extend the implementation of the ISOBASE node so that it can serve also as an ISOTOPE node? As evident from the diagram in Figure 3.7, it needs support for the `load_text` parameter. Its value must be sent along when calling `toplevel_spawn/2`, which will inject the code in the private database of the toplevel. Moreover, the goal identifier must be based on *both* the goal, the template and this value. Code for handling all of this would be easy to add.

### C.1.4 Implementing the Erlang-style concurrency predicates

This section implement specifications for how we believe predicates such as `spawn/1-3`, `exit/1-2`, `!/2` and `receive/1-2` might work. To keep things as succinct as possible we do not add code checking the instantiation of arguments. (However, some such tests are present in the proof-of-concept mini implementation.)

Today widely available Prolog systems can be differentiated whether they are multi-threaded or not. In a multi-threaded Prolog system we can create multiple threads that run concurrently over the same knowledge base. From Table 2 in *Fifty Years of Prolog and Beyond* we learn that out of the Prolog systems listed above, five implement multi-threading support. According to this table, these are Ciao, ECLiPSe, SWI-Prolog, tuProlog and XSB. However, we that that Trealla Prolog should also be added to the list, and thus we have six systems with multi-threading support.

There is a draft standard for multi-threading support in Prolog, specified in a document that begins like so:

> ISO/IEC DTR 13211–5:2007 Prolog multi-threading support [...] is an optional part of the International Standard for Prolog, ISO/IEC 13211. [...] Multi-thread predicates are based on the semantics of POSIX threads. They have been implemented in some Prolog systems. As such, they are deemed a worthy extension to the ISO/IEC 13211 Prolog standard.[8]

Except for Ciao Prolog, which takes a different approach to multi-threading, the six systems listed above all implement the draft standard.

In order to support the Erlang-style concurrency predicates offered by the ACTOR profile of Web Prolog the five predicates on the left can be implemented by means of the seven predicates from the draft standard on the right:

```
spawn/3                 thread_create/3
self/1                  thread_self/1
!/2, send/2             thread_send_message/2
receive/1-2             thread_get_message/3
exit/2                  thread_signal/2
                        thread_detach/1
                        thread_property/2
```

[8] `https://logtalk.org/plstd/threads.pdf`

The drafts standard specifies more than a dozen more predicates, such as predicates for creating message queues and managing mutexes. We do not need those.

Change start/4 into initialize/4, Changne stop/2 into terminate/2, change down/3 to terminate/3, see to it that eason 'kill' is converted into killed, by changing the implementation if down_reason/2?

Here is a first sketch of an implementation of `spawn/1-3`:

```
:- op(800,  xfx, !).
:- op(1000, xfy, when).

:- dynamic link/2.

spawn(Goal) :-
    spawn(Goal, _Pid).

spawn(Goal, Pid) :-
    spawn(Goal, Pid, []).

spawn(Goal, Pid, Options) :-
    thread_self(Self),
    make_pid(Pid),
    thread_create(start(Self, Pid, Goal, Options), Pid, [
        alias(Pid),
        at_exit(stop(Pid, Self))
    ]),
    thread_get_message(initialized(Pid)).


make_pid(Pid) :-
    random_between(10000000, 99999990, Num),
    atom_number(Pid, Num).


:- thread_local parent/1.

start(Parent, Pid, Goal, Options) :-
    assertz(parent(Parent)),
    option(link(Link), Options, true),
    (   Link == true
    ->  assertz(link(Parent, Pid))
    ;   true
    ),
    option(monitor(Monitor), Options, false),
    (   Monitor == true
    ->  assertz(monitor(Parent, Pid))
    ;   true
    ),
    thread_send_message(Parent, initialized(Pid)),
    call(Goal).
```

```
stop(Pid, Parent) :-
    thread_detach(Pid),
    retractall(link(Parent, Pid)),
    retractall(registered(_Name, Pid)),
    forall(retract(link(Pid, ChildPid)),
           exit(ChildPid, kill)),
    down_reason(Pid, Reason),
    forall(retract(monitor(Other, Pid)),
           Other ! down(Pid, Reason)).

down_reason(Pid, Reason) :-
    retract(exit_reason(Pid, Reason)),
    !.
down_reason(Pid, Reason) :-
    thread_property(Pid, status(Reason)).
```

All of the above must be replaced - especially down/2 must become down/3

A thread implements an actor. The thread comes with its own message queue, which will serve as the actor's mailbox. The thread identifier works like a pid.

A number of thread-related predicates are called that finds the identity of the soon-to-become parent, creates a thread that, just before terminating, calls `down/3`, which takes care of what must be done in the last moment before the actor terminates – the termination of any children that it may have spawned during its life cycle (in case `link` is set to `true`), and the sending of a `down` message to the parent (if `monitor` is set to `true`).

The above implementation of `spawn/1-3` calls two predicates – `exit/2` and `!/2` – that must be implemented. In addition, `exit/1` must be implemented, and this can be done as followos:

```
:- dynamic exit_reason/2.

exit(Reason) :-
    self(Self),
    asserta(exit_reason(Self, Reason)),
    abort.
```

For the implementation of `exit/2`, ISO/IEC DTR 13211–5:2007 specifies a predicate `thread_signal/2` to make a thread execute some goal as an interrupt. Signaling may be used to cancel no-longer-needed threads. This means that `exit/2` may be implemented like so:

```
exit(Pid, Reason) :-
    catch(thread_signal(Pid, exit(Reason)),
          error(existence_error(_,_), _),
          true).
```

Note that `thread_signal/2` throws an error if the thread ID in the first argument points to a thread that does not exist. Since `exit/2` must succeed also in this case, we have wrapped the call to `thread_signal/2` in a call to `catch/3`.

For the implementation of `!/2`, ISO/IEC DTR 13211–5:2007 offers a predicate `thread_send_message/2` which is somewhat similar to Erlang's send primitive. It allows any term to be sent to any thread. Just like in Erlang, the term is copied to the receiving process and variable bindings are thus lost. However, `thread_send_message/2` throws an error if the thread ID in the first argument points to a thread that does not exist. Again, since `!/2`, just like in Erlang, should succeed also in this case, we wrap the call in `catch/3` like so:

```
Pid ! Message :-
    send(Pid, Message).

send(Name, Message) :-
    registered(Name, Pid),
    !,
    send(Pid, Message).
send(Pid, Message) :-
    catch(thread_send_message(Pid, Message),
          error(existence_error(_,_), _),
          true).
```

In effect, this makes any attempt to send a message to a non-existing actor a no-op.

The implementation of the receive operation is somewhat more involved. Relying on `thread_get_message/3`, what might be regarded as a reference implementation of `receive/1-2` looks like this:

```
:- thread_local deferred/1.

receive(Clauses) :-
    receive(Clauses, []).

receive(Clauses, Options) :-
    thread_self(Mailbox),
    (   clause(deferred(Msg), true, Ref),
        select_body(Clauses, Msg, Body)
    ->  erase(Ref),
        call(Body)
    ;   receive(Mailbox, Clauses, Options)
    ).

receive(Mailbox, Clauses, Options) :-
    (   thread_get_message(Mailbox, Msg, Options)
    ->  (   select_body(Clauses, Msg, Body)
        ->  call(Body)
```

```
        ;   assertz(deferred(Msg)),
            receive(Mailbox, Clauses, Options)
        )
    ;   option(on_timeout(Body), Options, true),
        call(Body)
    ).

select_body(_M:{Clauses}, Message, Body) :-
    select_body_aux(Clauses, Message, Body).

select_body_aux((Clause ; Clauses), Message, Body) :-
    (   select_body_aux(Clause,  Message, Body)
    ;   select_body_aux(Clauses, Message, Body)
    ).
select_body_aux((Head -> Body), Message, Body) :-
    (   subsumes_term(if(Pattern, Guard), Head)
    ->  if(Pattern, Guard) = Head,
        subsumes_term(Pattern, Message),
        Pattern = Message,
        catch(once(Guard), _, fail)
    ;   subsumes_term(Head, Message),
        Head = Message
    ).
```

### C.1.5 Implementing output/1, input/2 and respond/2

The predicates `output/1-2`, `input/2-3` and `respond/2` are implemented on top of combinations of the `!/2` and `receive/2-3` primitives.

Here is the suggested implementation of `output/1-2`:

```
output(Term) :-
    output(Term, []).

output(Term, Options) :-
    self(Self),
    parent(Parent),
    option(target(Target), Options, Parent),
    Target ! output(Self, Term).
```

The implementation of `input/2-3` is slightly more complicated:

```
input(Prompt, Input) :-
    input(Prompt, Input, []).
```

```
input(Prompt, Input, Options) :-
    self(Self),
    parent(Parent),
    option(target(Target), Options, Parent),
    Target ! prompt(Self, Prompt),
    receive({
       '$input'(Target, Input) ->
           true
    }).
```

The predicate `respond/2` is used to respond to a prompt:

```
respond(Pid, Term) :-
    self(Self),
    Pid ! '$input'(Self, Term).
```

Since messages are wrapped in a binary term with the pid of the sender in its first argument, the target will always know if the message came from the right process.

### C.1.6 What is missing from the sketches?

The predicates implemented so far are sufficient for running the majority of the example programs given in Chapter 2 and Chapter 3 of this book. Of course, this is just a start, and to be able to run *all* programs, and in particular the ones in Chapter 4, more is needed. Notably, the current implementation sketch does not support

- network-transparent concurrency and distribution,
- the implementation of an actors's private database, and
- security.

As for network transparency, the scenarios in Chapter 4 show in great detail how the stateful distribution layer might work. Recall that to spawn an actor on a remote node, the `node` option must be passed to `spawn/3` with a URI pointing to the node:

```
?- spawn(foo, Pid, [
       node('http://n7.org')
   ]).
Pid = 34925412@'http://n7.org'.
?-
```

Note that once this works for `spawn/3`, it will work for `toplevel_spawn/2` too.

Exiting *remote* processes must also be implemented so that it can be used in the following way:

```
?- exit(34925412@'http://n7.org', normal).
true.
?-
```

Our implemention of the send operator will only work for the simplest of cases of local messaging, but a complete implementation of an `actor` node must also allow sending to remote processes, like so:

```
?- 34925412@'http://n7.org' ! bar.
true.
?-
```

Once this works for `!/2`, it will also make `toplevel_call/2-3`, `toplevel_next/1-2` and related predicates work.

Note that the stateful distribution layer depends on WebSockets and that, as far as we know, at this point in time SWI-Prolog is the only Prolog system that offers a WebSocket library.

Source code injection such as in the following example must also be supported by an ACTOR node:

```
?- spawn(baz, Pid, [
      load_text('p(a). p(b).')
   ]).
Pid = 71123976@'http://n1.org'.
?-
```

Injected source code must end up in the spawned actor's private Prolog database and thus we need a viable approach to the implementation of this database and the isolation it requires. Isolation can be based on `thread_local/1` or the use of temporary modules. (Temporary modules are used by `library(pengines)`.)

If source code injection works for `spawn/3`, it will work for `toplevel_spawn/2` and `rpc/3` as well.

On the subject of security, a very important requirement relates to *sandboxing*. The approach taken by `library(sandbox)` in SWISH is not satisfactory.

### Implementation on top of engines

Here is how the SWI-Prolog manual introduces engines, an idea that originated by Paul Tarau.

> Engines are closely related to threads. An engine is a Prolog virtual machine that has its own stacks and (virtual) machine state. Unlike normal Prolog threads though, they are not associated with an operating system thread. Instead, you ask an engine for a next answer (`engine_next/2`). Asking an engine for the next answer attaches the engine to the calling operating system thread and cause it to run until the engine calls `engine_yield/1` or its associated goal completes with an answer, failure or an exception. After the engine yields or completes, it is detached from the operating system thread and the answer term is made available to the calling thread. Communicating with an engine is similar to communicating with a Prolog system though the terminal. In this sense engines are related to Pengines as provided by library `library(pengines)`, but where Pengines aim primarily at accessing Prolog engines through the network, engines are in-process entities.

—

Some Prolog systems implement concurrency but do not use ISO/IEC DTR 13211–5:2007. SICStus Prolog and Ciao Prolog are cases in point, but there are others. Presumably, at least some of the Web Prolog profiles can still be implemented using alternative constructs provided by these dialects.[9]

## C.2 Sandboxing, isolation, and capability control

A production-quality Prolog node requires more than correct semantics and acceptable performance. It must also enforce a security boundary strong enough to allow the execution of untrusted client code. From an implementation point of view, this raises a practical question: where should that boundary be drawn, and by what mechanisms should it be enforced?

At least four concerns must be kept apart. First, there is *expressibility*: which language constructs and built-ins are available to client code. Second, there is *authority*: which actors, services, and external resources a computation may affect. Third, there is *isolation*: whether the runtime itself can escape into the host operating system or interfere with other computations. Fourth, there is *resource consumption*: how much CPU time, memory, process space, mailbox space, or network activity a session may consume. These concerns are related, but they are not identical, and no single mechanism addresses them all.

### C.2.1 Whitelist and blacklist strategies

A first implementation choice concerns the form of the language-level sandbox. One strategy is *whitelisting*: client code is allowed to call only predicates that have been explicitly declared safe. Another is *blacklisting*: client code is allowed to use the exposed language fragment except for specifically forbidden predicates or capabilities.

For a reflective Prolog system, whitelisting is the safer default. The reason is not merely that some predicates are dangerous, but that authority may often be reconstructed indirectly through meta-calling, predicate inspection, dynamic database operations, foreign predicates, or other reflective mechanisms. In such an environment, whitelisting has a clear advantage: newly added predicates are denied by default until they have been reviewed.

Blacklisting becomes more plausible only when the language fragment exposed to clients is already narrow and capability-oriented. This is one reason why Web Prolog is a better candidate for blacklisting than a full Prolog system. If client code is already denied direct access to file I/O, operating-system interaction, unrestricted reflection, foreign predicates, and persistent code modification, then the remaining

[9] (Clark et al., 2001), section 7 compare communication in Qu-Prolog with that of SICStus-MT, BinProlog, CIAO, Erlang, Mozart-Oz and April.

surface is small enough that one may hope to enumerate the forbidden capabilities rather than all the permitted predicates. Even then, however, blacklisting should be treated as an engineering convenience rather than as a foundation for trust.

### C.2.2 Language sandboxing versus host isolation

Language-level sandboxing is only one layer of defence. Even a carefully restricted Prolog fragment still runs inside some host environment, and that host may itself provide dangerous authority. For this reason, a node implementation should distinguish between *language sandboxing* and *host isolation*.

Language sandboxing decides what client code can express in Web Prolog. Host isolation decides what the runtime process itself is able to do. A system that relies only on sandboxed predicates, but runs with unrestricted filesystem and network access in the host process, still places great trust in the correctness of the sandbox implementation. Conversely, a strongly isolated host environment can remove whole classes of risk even if the Prolog-level sandbox is imperfect.

This suggests a layered implementation strategy. The inner layer is the Web Prolog language fragment exposed to clients. Around that sits a capability policy governing which exported services, actors, and owner-defined predicates may be reached. Outside that sits the deployment environment, which should itself restrict what the runtime may access.

### C.2.3 WebAssembly and WASI

A particularly interesting host environment for Web Prolog is WebAssembly. A WebAssembly runtime starts with substantially less ambient authority than a native process. In the browser it inherits the browser's security model; in a standalone embedding it can be run inside a tightly controlled execution environment.

The case becomes even stronger with WASI. In a WASI-style setting, access to external resources is explicitly granted rather than silently inherited. For a Web Prolog implementation, this means that many operations that would otherwise need to be denied at the Prolog level may simply be absent from the host environment. If the runtime is not given filesystem handles, unrestricted sockets, or privileged host callbacks, then no client program can reach them, regardless of whether the language-level sandbox is phrased as a whitelist or a blacklist.

This does not solve everything. Wasm and WASI do not by themselves control Prolog-level reflection, and they do not remove the need for quotas and timeouts. But they can substantially reduce the amount of trust that must be placed in the Prolog sandbox alone.

### C.2.4 Capability control inside the node

Even inside a host-isolated runtime, a Prolog node still needs a policy for authority within the Prolog Web. An implementation must decide which operations are available to ordinary clients and which are reserved for node owners. This includes, at minimum, the distinction between session-local computation and durable node-level authority.

Examples of owner-only capabilities include publishing services, registering public names, installing persistent code, modifying shared node state, or detaching computations so that they survive the session that created them. Client capabilities, by contrast, should normally be limited to session-scoped actor creation, messaging, remote procedure calls to exported services, and other operations whose effects are contained by the lifetime discipline described earlier in the book.

Implementation-level security is therefore not merely a matter of preventing operating-system escape. It is equally a matter of ensuring that the node's own public surface is small, explicit, and role-sensitive.

### C.2.5 Resource limits are not sandboxing

It is important to distinguish *sandboxing* from *resource governance*. Timeouts, actor limits, mailbox bounds, RPC limits, and cache bounds are essential to a production deployment, but they serve a different purpose. A sandbox answers the question *what may this program do?* Resource governance answers the question *how much may it do?*

This distinction matters in implementation work. A program may remain entirely within the sandbox and still cause denial-of-service by spawning large numbers of actors, generating very large terms, forcing unbounded result streams, or repeatedly issuing remote calls. Conversely, a program may use very few resources and yet still violate the security boundary if it reaches an authority it should not possess. A complete node therefore requires both capability control and operational limits.

### C.2.6 A realistic implementation strategy

For near-term implementations, the most realistic strategy is therefore a layered one. The exposed Web Prolog fragment should be restricted by a language-level sandbox. The node should enforce a capability policy distinguishing client authority from owner authority. Sessions should be subject to operational limits. And, where possible, the runtime should be deployed inside a host environment that reduces ambient authority, such as a browser sandbox, a Wasm runtime, a WASI-based embedding, or an operating-system-level isolation boundary.

Under such a strategy, whitelisting and blacklisting become local design choices inside a broader architecture of defence. For implementations built directly on top of existing Prolog systems, whitelisting is likely to remain the safer option. For restricted Web Prolog runtimes deployed inside capability-oriented hosts such as Wasm or WASI, blacklisting may become more plausible because so much dangerous authority has already been removed at the outer layers.

## C.3 Alternative implementation approaches

Some (most?) Prolog systems could probably be deployed on the Web using OS based sandboxing where every client talks to a private copy. There are several language neutral environments that would let us do this. At least some profiles of Web Prolog could probably be implemented using this approach.

### C.3.1 Implementation for browsers

There seems to be at least three approaches one might take in order to implement a Web Prolog system (minus features that makes it into a node) that runs in a browser:

1. **JavaScript implementation:** Prolog can be interpreted or compiled into JavaScript, allowing it to run directly in a web browser. JavaScript engines are highly optimized and available in all modern browsers, providing a convenient platform. However, the performance and capabilities might be limited compared to native Prolog environments. This approach is taken by Tau Prolog.[10]
2. **WebAssembly (WASM):** WebAssembly offers a way to run code written in multiple languages at near-native speed in web browsers. Some Prolog system can be compiled to WebAssembly, enabling it to run efficiently in the browser. This approach could potentially offer a better performance than a pure JavaScript implementation. This approach is taken by SWI-Prolog, Ciao Prolog, Trealla Prolog and Scryer Prolog.
3. **Browser extensions or plug-ins:** Developing a browser extension or plug-in could be another way to integrate Prolog into a web environment. This would allow more direct interaction with the browser's capabilities but might limit the audience due to the need for installing the extension.

There is actually a fourth approach: to try to convince browser providers (e.g. Mozilla) that they should support Prolog. Interestingly, already back in 1999, Mozilla ran an umbrella project for experimental work investigating the integration of logic and inference capabilities into the RDF based information management system in

[10] `http://tau-prolog.org/`

the Mozilla application environment. Based on SWI-Prolog, Geoff Chappel implemented an early prototype Prolog plugin for the Mozilla/Netscape browser.[11] Presumably, the same approach can be used to support Prolog in Firefox. The problem with this approach is that it would likely be very hard to convince other browser vendors to follow suit.

—

As suggested in ??, the most promising approach to enable the running of Web Prolog in browsers seems to be using WebAssembly.

—

Flach and Wielemaker: ”There are two ways to achieve that. One is to run Prolog in a WebWorker (roughly, a browser task without a window) and make it communicate with the browser task associated to a normal browser window. The other is to yield from the Prolog virtual machine and resume Prolog when the data become available. This leads to at least two interface designs: – with a WebWorker we get interaction with a Prolog process that is not very different from Prolog running on a remote server.”

—

Problems: No concurrency.

—

Trealla Prolog:
https://news.ycombinator.com/item?id=35624654
https://github.com/trealla-prolog/trealla

### C.3.2 Implementations on top of other programming systems

Dedicated Prolog systems such as SWI-Prolog or SICStus Prolog are usually written in fairly low-level languages such as C, C++ or (more recently) Rust, but one surprisingly often also comes across Prolog implementations in more high-level languages such as Scala, Haskell, Erlang, Lisp or Scheme.

Such implementations may be nothing more than the result of programming exercises – perhaps by a student or a hobbyist wishing to implement a couple of somewhat challenging algorithms such as unification and backtracking. But they may also be the results of a real need felt by a programmer confronted with a problem – the need for a “rule engine” or “logic engine.” Then, rather than move an application to Prolog, the programmer decides to implement Prolog in the language of his application. Such implementations are often slow and incomplete, but there are exceptions. Indeed, it can be very useful to embed Prolog-style unification and backtracking in a more traditional imperative or functional programming language. Some people even do it for fun.

[11] `https://www-archive.mozilla.org/rdf/doc/inference.html`

Interestingly, there is already Erlog[12] – an unusually complete Prolog implementation in Erlang written by Robert Virding (one of the people behind Erlang, who seems to belong to the do-i-for-fun category of programmers).

An implementation of a Web Prolog node in Erlang would be interesting as it would most probably have a performance profile very different from the implementation in SWI-Prolog. Prolog is likely to be a lot slower in Erlang than in the implementation in SWI-Prolog, whereas the super-fast lightweight processes in Erlang have other advantages, making it scale better to very many simultaneous users on a network. Besides, Erlang is particularly famous for extremely efficient implementations of web-related technologies such as web servers (e.g. Yaws and Cowboy) and this could also be a distinctive advantage of an Erlang implementation.

Then there is Akka + (say) tuProlog.

### C.3.3 Prospects for a dedicated Web Prolog VM

The Erlang virtual machine is known as the BEAM.[13] The BEAM VM is the virtual machine at the core of the Erlang Open Telecom Platform (OTP), designed to efficiently execute Erlang code. The BEAM VM is renowned for its support of highly concurrent, distributed, and fault-tolerant applications, qualities that stem directly from the design principles of the Erlang language itself. One of the defining characteristics of the BEAM VM is its lightweight process model. Unlike operating system processes or threads, BEAM processes are managed within the VM, allowing for the creation of millions of concurrent processes. These processes are isolated, communicate via message passing, and do not share memory, which mitigates common concurrency issues such as race conditions and deadlocks.

Compare performance of Erlang vs Web Prolog on the process ring.

The BEAM VM is not limited to running Erlang code. It also supports other languages that compile to its bytecode, such as Elixir, a modern, functional programming language that is fully interoperable with Erlang, leveraging the same VM for concurrent, distributed, and fault-tolerant applications.

In a paper presented at an Erlang conference ??, the following statement was made:

> The holy grail for a Web Prolog runtime system is a compiler targeting a virtual machine with BEAM-like properties, capable of producing code which when run will create processes as small and efficient as Erlang processes, yet with the useful capabilities that Prolog offers. *We do not dare to guess whether building such a virtual machine is feasible*.

When Richard O'Keefe, with a lot of experience with both Prolog and Erlang, kindly agreed to read the paper, he made the following comment on the passage:

> I do! It is! Let's face it, processes in Logix were *tiny* compared with Erlang ones. And the BEAM was a reaction to JAM, which was inspired by the WAM, and Aquarius Prolog used the BAM which had some similarities to BEAM.

[12] `https://github.com/rvirding/erlog`

[13] `http://blog.erlang.org/beam-compiler-history`

Designing and implementing a BEAM-like VM for Web Prolog is likely to be really hard, but with time it might prove to be the optimal way forward. One must not forget, however, that a node built on top of a current mature Prolog system might be able to offer a client a lot more than an efficient but bare-bones implementation which compiles source code into a BEAM-like VM.

### C.3.4 Less capable profiles need less work

Unfortunately, implementing a Prolog Web node capable of running the full set of example programs given in the previous chapters is not exactly an easy task, and this is what is required of an ACTOR node. Such difficulties have motivated us to define three other profiles which are less powerful and easier to implement than the ACTOR profile. Hopefully, this may encourage developers of Prolog systems to implement nodes capable of contributing to the Prolog Web.

An ISOBASE node can, since it does not have to load data or programs dynamically, pull a few tricks in order to optimise query processing, tricks that may be slow to play, but which can be performed offline if dynamic loading is not permitted. (Compiling Web Prolog code into C, for example.) In contrast, a node that implements the ACTOR or ISOTOPE profile, and thus offers code injection capabilities, must do the updating quickly and must probably avoid some of the slower ways of optimizing code.

Without doubt, a node that conforms to the RELATION profile would be the easiest to implement, especially since it does not have to implement any ISO Prolog built-in predicates or control structures. One can imagine writing a RELATION node in (say) Python which offers developers just a single owner-defined predicate. If it supports querying over the stateless HTTP API in the manner described elsewhere in this book, and is able to return answers in the form of either Prolog or JSON, then it it does conform to the RELATION profile of Web Prolog, yet would be easy to implement. Note that by our own definition, since it is a process on the Prolog Web that talks Prolog, such a node must also be categorized as a Prolog agent, despite not being written in Prolog.

# Appendix D
# Benchmarking

> And then we've got this sort of dichotomy between efficiency and clarity. You know, to make something clearer, you add a layer of abstraction and to make it more efficient you remove a layer of abstraction. So go for the clarity bit. Wait ten years and it will be a thousand times faster, you want it a million times faster, wait 20 years.
>
> *Joe Armstrong*

All benchmarks were run on a 2023 Apple iMac with the M3 chip and 16 GB of RAM.

## D.1 Benchmarking spawn

In their 2009 textbook, Cesarini and Thompson present a way to benchmark process creation and message passing in Erlang. Below we are going to use a translation into Web Prolog of their benchmarking code and see what kind of performance we can expect from Web Prolog and compare it with the performance of Erlang.

In this benchmark, "the parent spawns a child and sends a message to it. Upon being spawned, the child creates a new process and waits for a message from its parent. Upon receiving the message, it terminates normally. The child's child creates yet another process, resulting in hundreds, thousands, and even millions of processes."

Here is the program in Web Prolog:

```
start(Num) :-
    self(Self),
    start_proc(Num, Self).

start_proc(0, Pid) :- !,
```

```
    Pid ! ok.
start_proc(Num, Pid) :-
    Num1 is Num-1,
    spawn(start_proc(Num1, Pid), NPid),
    NPid ! ok,
    receive({ok -> true}).
```

Here is the result of spawning 100,000 and 1,000,000 processes:

```
?- time(start(100 000)).
 % 41 inferences, 0.000 CPU in 23.918 seconds
true.
?- time(start(1 000 000)).
 % 41 inferences, 0.001 CPU in 251.514 seconds
true.
?-
```

Here is the original Erlang program:

```
-module(benchmark).
-export([start/1, start_proc/2]).

start(Num) ->
    start_proc(Num, self()).

start_proc(0, Pid) ->
    Pid ! ok ;
start_proc(Num, Pid) ->
    NPid = spawn(?MODULE, start_proc, [Num-1, Pid]),
    NPid ! ok,
    receive ok -> ok end.
```

Here is the result of spawning 100,000 and 1,000,000 processes:

```
1> c(benchmark).
{ok,benchmark}
2> timer:tc(benchmark, start, [100000]).
{89440,ok}
3> timer:tc(benchmark, start, [1000000]).
{644545,ok}
```

Note that times are given in microseconds, which means that it took 0.09 seconds to spawn 100,000 processes, and 0.64 seconds to spawn 1,000,000 processes.

It means that Erlang is around an impressively 250 times faster than Web Prolog when spawning 100,000 processes, and around 400 times faster when spawning 1,000,000 processes. (We estimate that we would have to wait ten years to reach the performance of present day Erlang.)

## D.2 Benchmarking send and receive

One of the examples in Chapter 2 is a program featuring two actors playing ping-pong. bm_ping_pong/1 is a version of this program suitable for benchmarking send and receive where output has been eliminated:

```
bm_ping(0, Pong_Pid) :-
    Pong_Pid ! finished.
bm_ping(N, Pong_Pid) :-
    self(Self),
    Pong_Pid ! ping(Self),
    receive({
        pong -> true
    }),
    N1 is N - 1,
    bm_ping(N1, Pong_Pid).

bm_pong :-
    receive({
        finished -> true;
        ping(Ping_Pid) ->
            Ping_Pid ! pong,
            bm_pong
    }).

bm_ping_pong(N) :-
    spawn(bm_pong, Pong_Pid),
    spawn(bm_ping(N, Pong_Pid), Ping_Pid, [
        monitor(true)
    ]),
    receive({
       down(_, Ping_Pid, true) ->
          true
    }).
```

Here is the result of running it with N=100,000 and N=1,000,000:

```
?- time(bm_ping_pong(100 000)).
 % 39 inferences, 0.000 CPU in 0.839 seconds
true.
?- time(bm_ping_pong(1 000 000)).
 % 39 inferences, 0.000 CPU in 8.266 seconds
true.
?-
```

Here is the same program in Erlang:

```
-module(bm_pingpong).
-export([bm_start/1, bm_ping/2, bm_pong/0]).

bm_ping(0, Pong_PID) ->
    Pong_PID ! finished ;

bm_ping(N, Pong_PID) ->
    Pong_PID ! {ping, self()},
    receive
        pong -> true
    end,
    bm_ping(N - 1, Pong_PID).

bm_pong() ->
    receive
        finished -> true ;
        {ping, Ping_PID} ->
            Ping_PID ! pong,
            bm_pong()
    end.

bm_start(N) ->
    Pong_PID = spawn(bm_pingpong, bm_pong, []),
    spawn_monitor(bm_pingpong, bm_ping, [N, Pong_PID]),
    receive Msg -> ok end.
```

Here is the result running it with `N=100,000` and `N=1,000,000`:

```
2> timer:tc(pingpong, start, [100000]).
{80930,ok}
3> timer:tc(pingpong, start, [1000000]).
{554502,ok}
4>
```

Erlang is again faster than Web Prolog, although the difference is less here than when spawning processes. Erlang is around 10 times faster than Web Prolog for `N=100,000`, and around 15 times faster for `N=1,000,000`.

## D.3 Benchmarking rpc/2-3 running over HTTP

We use a Prolog file with Wordnet data,[1] but only the predicate `s/6` from the Wordnet file `wn_s.pl` and define two predicates, `word/2` and `type/2`, like so:

[1] https://wordnet.princeton.edu/download/current-version.

```
word(S, Word) :- s(S, _, Word, _, _, _).

type(S, Type) :- s(S, _, _, Type, _, _).
```

Here are timings for some queries. Also shows how many solutions there are:

```
% How many words are there in Wordnet?
?- time(findall(.,word(S,_), L)), length(L, N).
 % 212,567 inferences, 0.033 CPU in 0.037 seconds
N = 212556.
?-

% How many verbs are there in Wordnet?
?- time(findall(.,type(S,v), L)), length(L, N).
 % 25,058 inferences, 0.004 CPU in 0.004 seconds
N = 25047.
?-
```

Here is the timing for solving the query `?-type(S,v),word(S,_)` wrapped in `rpc/2`:

```
?- URI = 'http://localhost:3010',
   time(findall(.,rpc(URI,(type(S,v),word(S,_))),L)), length(L, N).
 % 150,283 inferences, 0.071 CPU in 0.148 seconds
N = 74201.
?-
```

So that seems to be efficient enough. But what we really want is a query that does searches on two different nodes. We will use only one computer on which two nodes are running, `http://localhost:3010` and `http://localhost:3011`.

As we can see below, performance is absolutely no problem if we want to look at the solutions one by one in a terminal:

```
?- time((rpc('http://localhost:3010', type(S,v)),
         rpc('http://localhost:3011', word(S,W)))).
 % 5,060 inferences, 0.010 CPU in 0.029 seconds
S = 200001740,
W = breathe ;
 % 4 inferences, 0.000 CPU in 0.000 seconds
S = 200001740,
W = 'take a breath' ;
 % 2 inferences, 0.000 CPU in 0.000 seconds
S = 200001740,
W = respire ;
 % 2 inferences, 0.000 CPU in 0.000 second
S = 200001740,
W = suspire ;
...
```

That is not much of a benchmark, though. Let us instead measure how long it takes to compute all 74201 solutions:

```
?- _URI1 = 'http://localhost:3010',
   _URI2 = 'http://localhost:3011',
   time(forall((rpc(_URI1, type(S,v)),
                rpc(_URI2, word(S,W))),true)).
 % 50,349,704 inferences, 3.461 CPU in 73.723 seconds
true.
?-
```

That is fairly slow.[2] The main reason is that since the first occurrence of the call to `rpc/2` has 25047 solutions, the second occurrence of `rpc/2` is *called* 25047 times!

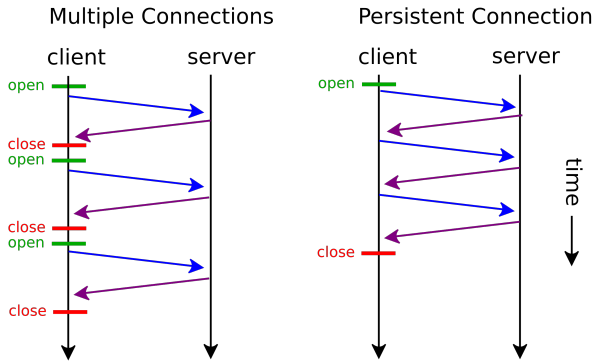



**Fig. D.1** Persistent connections.

Passing the option `connection('Keep-alive')` to the second call to `rpc/2-3` leads to a significant improvements:

```
?- _URI1 = 'http://localhost:3010',
   _URI2 = 'http://localhost:3011',
   _Opts = [connection('Keep-alive')],
   time(forall((rpc(_URI1, type(S,v)),
                rpc(_URI2, word(S,W), _Opts)),true)).
 % 53,811,383 inferences, 2.912 CPU in 7.170 seconds
true.
?-
```

The following query confirms that making 25047 network roundtrips to `localhost` takes around 45 seconds, so by far that's where the most time is spent:

[2] We are a bit suspicious of this result. See `https://swi-prolog.discourse.group/t/web-prolog-isobase-test-cluster-hack-away-find-security-bugs/1425/96wherebetterperformancewasachieved,onaslowercomputer`.

```
?- URI = 'http://localhost:3010',
   time(forall((between(1,25047,_), rpc(URI, true)),true)).
 % 43,080,839 inferences, 6.803 CPU in 45.369 seconds
true.
?-
```

Note that `rpc/2` was used here, which meant that the default `limit=none` was used. As we see here, and as expected, setting limit to a lesser value such as `100` is even slower:

```
?- URI1 = 'http://localhost:3010',
   URI2 = 'http://localhost:3011',
   time(forall((rpc(URI1, type(S,v), [limit(100)]),
                rpc(URI2, word(S,W), [limit(100)])),true)).
 % 44,963,746 inferences, 7.533 CPU in 62.859 seconds
true.
?-
```

Setting limit to 1 is the slowest:

```
?- URI1 = 'http://localhost:3010',
   URI2 = 'http://localhost:3011',
   time(forall((rpc(URI1, type(S,v), [limit(1)]),
                rpc(URI2, word(S,W), [limit(1)])), true)).
 % 225,593,720 inferences, 29.619 CPU in 244.431 seconds
true.
?-
```

# Appendix E
# Learning material

## E.1 Books teaching Prolog

- *The Art of Prolog*.
- Patrick Blackburn, Johan Bos, and Kristina Striegnitz. *Learn Prolog Now! An introduction to Prolog programming.*
  `http://www.learnprolognow.org/lpnpage.php`
- Michael A. Covington, Donald Nute, and André Vellino. *Prolog Programming in Depth*, Second edition, Prentice-Hall. 1997.
  `http://www.covingtoninnovations.com/books/PPID.pdf`
- Michael A. Covington. *Natural Language Processing for Prolog Programmers.* Prentice-Hall, 1994.
  `http://www.covingtoninnovations.com/books/NLPPP.pdf`
- Bart Demoen, Phuong-Lan Nguyen, Tom Schrijvers, Remko Tronçon. *The First 10 Prolog Programming Contests*.
  `https://dtai.cs.kuleuven.be/ppcbook`
- Peter Flach, *Simply Logical: Intelligent Reasoning by Example*, John Wiley 1994.
  `http://www.cs.bris.ac.uk/~flach/SimplyLogical.html`
- Gerald Gazdar and Chris Mellish. *Natural Language Processing in Prolog*.
  `https://tinyurl.com/p459746`
- Dennis Merritt. *Building Expert Systems in Prolog*.
  `http://www.amzi.com/ExpertSystemsInProlog`
- Dennis Merritt. *Adventure in Prolog*.
  `http://www.amzi.com/AdventureInProlog`
- Fernando C. N. Pereira and Stuart M. Shieber. *Prolog and Natural-Language Analysis*.
  `http://mtome.com/Publications/PNLA/pnla.html`
- Markus Triska. *The Power of Prolog*.

- David Poole and Alan Mackworth. *Artificial Intelligence: Foundations of computational agents*.
  https://artint.info/2e/html/ArtInt2e.html

## E.2 Books teaching Erlang

- *Programming Erlang* from Pragmatic.
  https://www.pragprog.com/titles/jaerlang2/programming-erlang-2nd-edition/
- *Learn You Some Erlang for Great Good!* from No Starch Press.
  https://www.nostarch.com/erlang
- *Erlang Programming* from O'Reilly.
  https://www.oreilly.com/library/view/erlang-programming/9780596803940
- *Introducing Erlang* from O'Reilly.
  https://www.oreilly.com/library/view/introducing-erlang-2nd/9781491973363/
- *Erlang and OTP in Action* from Manning.
  https://www.manning.com/logan
- *Designing for Scalability with Erlang/OTP* from O'Reilly.
  https://shop.oreilly.com/product/0636920024149.do

# References


Arias J, Carro M, Salazar E, Marple K, Gupta G (2018) Constraint answer set programming without grounding. Theory and Practice of Logic Programming 18(3–4):337–354, DOI 10.1017/S1471068418000285

Armstrong J (2003a) Concurrency oriented programming in erlang. Invited talk, FFG

Armstrong J (2003b) Making reliable distributed systems in the presence of software errors. Phd thesis, Royal Institute of Technology, Stockholm

Armstrong J (2007) A history of erlang. In: Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III), San Diego, California, USA, 9-10 June 2007, pp 1–26, DOI 10.1145/1238844.1238850, URL https://doi.org/10.1145/1238844.1238850

Armstrong J, Virding R, Williams M (1995) Use of prolog for developing a new programming language. Appears in unlisted journal/magazine - verify source

Barnett J, Akolkar R, Auburn R, Bodell M, Burnett D, Carter J, McGlashan S, Lager T, Helbing M, Hosn R, Reifenrath K, Rosenthal N, Roxendal J (2015) State chart xml (scxml) state machine notation for control abstraction. w3c recommendation

Berners-Lee T, Hendler J, Lassila O (2001) The semantic web. Scientific American 284(5):34–43, URL http://www.sciam.com/article.cfm?articleID=00048144-10D2-1C70-84A9809EC588EF21

Brusk J, Lager T (2008) Developing natural language enabled games in SCXML. Journal of Advanced Computational Intelligence and Intelligent Informatics (JACIII) 12(2):156–163, DOI 10.20965/jaciii.2008.p0156, URL https://doi.org/10.20965/jaciii.2008.p0156

Ceri S, Gottlob G, Tanca L (1989) What you always wanted to know about Datalog (and never dared to ask). IEEE Transactions on Knowledge and Data Engineering 1(1):146–166, DOI 10.1109/69.43410

Cesarini F, Thompson S (2009) ERLANG Programming, 1st edn. O'Reilly Media, Inc.

Cicirelli F, Furfaro A, Giordano A, Nigro L (2009) Statechart-based actors for modelling and distributed simulation of complex multi-agent systems. In: European Conference on Modelling and Simulation, ECMS 2009, Madrid, Spain,

June 9-12, 2009, pp 233–239, DOI 10.7148/2009-0233-0239, URL https://doi.org/10.7148/2009-0233-0239

Clark K, Robinson PJ, Hagen R (2001) Multi-threading and message communication in qu-Prolog. Theory and Practice of Logic Programming 1:283–301

Clark KL, Gregory S (1986) PARLOG: Parallel programming in logic. ACM Transactions on Programming Languages and Systems 8(1):1–49, DOI 10.1145/5001.5390

Costantini S, Tocchio A (2004) The DALI logic programming agent-oriented language. In: Lifschitz V, Niemelä I (eds) Logics in Artificial Intelligence, Lecture Notes in Artificial Intelligence, vol 3229, Springer, pp 685–688, DOI 10.1007/978-3-540-30227-8_57

De Raedt L, Kimmig A, Toivonen H (2007) ProbLog: A probabilistic Prolog and its application in link discovery. In: Veloso MM (ed) Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI), pp 2462–2467

Gelernter D (1985) Generative communication in Linda. ACM Transactions on Programming Languages and Systems 7(1):80–112, DOI 10.1145/2363.2433

Green TJ, Huang SS, Loo BT, Zhou W (2013) Datalog and recursive query processing. Foundations and Trends in Databases 5(2):105–195, DOI 10.1561/1900000017

Hachey G, Gasevic D (2012) Semantic web user interfaces: A systematic mapping study and review

Harel D (1987) Statecharts: A visual formalism for complex systems. Sci Comput Program 8(3):231–274, DOI 10.1016/0167-6423(87)90035-9, URL https://doi.org/10.1016/0167-6423(87)90035-9

Harel D, Naamad A (1996) The statemate semantics of statecharts. ACM Trans Softw Eng Methodol 5:293–333, DOI 10.1145/235321.235322

Hebert F (2013) Learn You Some Erlang for Great Good!: A Beginner's Guide. No Starch Press, San Francisco, CA, USA

Hendler J (2001) Agents and the semantic web. IEEE Intelligent systems 16(2):30–37

Horrocks I (1999) Constructing the user interface with statecharts. Addison-Wesley Longman Publishing Co., Inc.

Horrocks I, Parsia B, Patel-Schneider P, Hendler J (2005) Semantic web architecture: Stack or two towers? In: Fages F, Soliman S (eds) Principles and Practice of Semantic Web Reasoning: Third International Workshop, PPSWR 2005, Dagstuhl Castle, Germany, September 11–16, 2005, Proceedings, Springer Berlin Heidelberg, Berlin, Heidelberg, pp 37–41

Junger D (2014) Client-side state-based control for multimodal user interfaces. Master's thesis, University of Gothenburg

Kakas AC, Kowalski RA, Toni F (1992) Abductive logic programming. Journal of Logic and Computation 2(6):719–770, DOI 10.1093/logcom/2.6.719

Kifer M, de Bruijn J, Boley H, Fensel D (2005) A realistic architecture for the semantic web. In: Adi A, Stoutenburg S, Tabet S (eds) Rules and Rule Markup Languages for the Semantic Web: First International Conference, RuleML 2005, Galway, Ireland, November 10–12, 2005. Proceedings, Springer Berlin Heidelberg, Berlin, Heidelberg, pp 17–29

Kowalski RA, Sadri F (2015) Reactive computing as model generation. New Generation Computing 33(1):33–67, DOI 10.1007/s00354-015-0103-z

Lager T, Myrendal J (2012) Exploring the web of coined catchy phrases. In: Proceedings of WWW2012: Web Science Track

Loke SW (2006) Declarative programming of integrated peer-to-peer and web based systems: the case of prolog. Journal of Systems and Software 79(4):523–536, URL http://dx.doi.org/10.1016/j.jss.2005.04.005

Lombardi A (2015) WebSocket: Lightweight Client-server Communications. O'Reilly Media, Inc.

Merritt D (1989) Building Expert Systems in Prolog. Springer-Verlag

Miller MS (2006) Robust composition: Towards a unified approach to access control and concurrency control. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, URL http://erights.org/talks/thesis/markm-thesis.pdf

Miller MS, Yee KP, Shapiro JS (2003) Capability myths demolished. Tech. Rep. SRL2003-02, Johns Hopkins University Systems Research Laboratory, URL https://srl.cs.jhu.edu/pubs/SRL2003-02.pdf

Muggleton S, De Raedt L (1994) Inductive logic programming: Theory and methods. The Journal of Logic Programming 19–20(Supplement 1):629–679, DOI 10.1016/0743-1066(94)90035-3

O'Keefe RA (1990) The Craft of Prolog. The MIT Press, Cambridge, MA

Ousterhout JK (1998) Scripting: higher level programming for the 21st century. Computer 31(3):23–30

Radomski S, Schnelle-Walka D, Radeck-Arneth S (2013) A prolog datamodel for state chart xml. In: Proceedings of the SIGDIAL 2013 Conference, Association for Computational Linguistics, pp 127–131, URL http://aclweb.org/anthology/W13-4019

Riguzzi F (2018) Foundations of Probabilistic Logic Programming: Languages, Semantics, Inference and Learning. River Publishers, Gistrup, Denmark, DOI 10.1201/9781003338192

van Roy P (2009) Programming paradigms for dummies: What every programmer should know. New computational paradigms for computer music 104

van Roy P, Haridi S (2004) Concepts, Techniques, and Models of Computer Programming. The MIT Press, Cambridge, Massachusetts

van Roy P, Haridi S, Brand P, Smolka G, Mehl M, Scheidhauer R (1997) Mobile objects in Distributed Oz. ACM Transactions on Programming Languages and Systems 19(5):804–851, DOI 10.1145/265943.265972

van Roy P, Brand P, Duchier D, Haridi S, Henz M, Schulte C (2003) Logic programming in the context of multiparadigm programming: The Oz experience. Theory and Practice of Logic Programming 3(6):715–763, DOI 10.1017/S1471068403001741

Russell SJ, Norvig P (2016) Artificial intelligence: a modern approach. Malaysia; Pearson Education Limited,

Samek M (2002) Practical statecharts in C/C++: Quantum programming for embedded systems. CRC Press

Skantze G, Moubayed SA (2012) Iristk: a statechart-based toolkit for multi-party face-to-face interaction. In: International Conference on Multimodal Interaction, ICMI '12, Santa Monica, CA, USA, October 22-26, 2012, pp 69–76, DOI 10.1145/2388676.2388698, URL https://doi.org/10.1145/2388676.2388698

Smolka G (1995) The Oz programming model. In: van Leeuwen J (ed) Computer Science Today: Recent Trends and Developments, Lecture Notes in Computer Science, vol 1000, Springer, Berlin, pp 324–343, DOI 10.1007/BFb0015252

van Steen M, Tanenbaum AS (2023) Distributed Systems, 4th edn. distributed-systems.net, free digital edition available from distributed-systems.net

Sterling L, Shapiro EY (1994) The Art of Prolog: Advanced Programming Techniques, 2nd edn. The MIT Press, Cambridge, MA

Svensson H, Fredlund LÅ, Earle CB (2010) A unified semantics for future erlang. In: Fritchie SL, Sagonas KF (eds) Proceedings of the 9th ACM SIGPLAN Workshop on Erlang, Baltimore, Maryland, USA, September 30, 2010, ACM, pp 23–32, URL http://dl.acm.org/citation.cfm?id=1863509

Tarau P (2000) Fluents: A refactoring of prolog for uniform reflection and interoperation with external objects. In: Lloyd JW, et al. (eds) Computational Logic - CL 2000, First International Conference, London, UK, 24-28 July, 2000, Proceedings, Springer, Lecture Notes in Computer Science, vol 1861, pp 1225–1239, DOI 10.1007/3-540-44957-4_82, URL https://doi.org/10.1007/3-540-44957-4_82

Van Gelder A, Ross KA, Schlipf JS (1991) The well-founded semantics for general logic programs. Journal of the ACM 38(3):620–650, DOI 10.1145/116825.116838

Virding R, Wikström C, Williams M, Armstrong J (1996) Concurrent programming in ERLANG (2nd ed.). Prentice Hall International (UK) Ltd., GBR

Wielemaker J, Beek W, Hildebrand M, van Ossenbruggen J (2016) Cliopatria: A swi-prolog infrastructure for the semantic web. Semantic Web 7(5):529–541

# Index