Torbjörn Lager

The Prolog Trinity ecosystem

Extracts from the current manuscripts

November 4, 2024

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 hopefully soon to become niche in the truly gigantic 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 implementors are indeed among its intended readers. However, we do not think a reader has to be a very experienced Prolog programmer or implementor 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 Erlang 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-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 work to the world, and in particular to other Web Prolog programmers. Oh, and we

[1] For readers lacking the necessary background in Prolog, we have a few recommendations to make in Appendix B.1.6.

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 believe, as we do, that it is likely 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 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. However, having said that, it is of course impossible to ignore the recent success of so called Large Language Models (LLMs), so we will speculate a little over the relation between LLM technology and our own approach to symbolic AI on the Web. We can only scratch the surface, but we 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.

Contents

Chapter 1
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*

1.1 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 first-order 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 was the husband of Pythias:

```
?- husband(pythias, aristotle).
true.
?-
```

Or we can ask who was the husband of Pythias:

```
?- husband(pythias, Husband).
Husband = aristotle.
?-
```

Only one solution was found (so it appears that Pythias was not a bigamist, and nor was Aristotle):

```
?- husband(Wife, aristotle).
Wife = pythias.
?-
```

Enumerate the married couples one by one!

```
?- 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.

Prolog allows the use of complex terms in the arguments of clauses, here in the form of *lists* in the well-known definition of `append/3`:

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

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 relation between the married couples, or the 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 **??**, 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]

1.2 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

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

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 1.1 shows a "cloud" of built-in control constructs and predicates specified by the ISO standard.

Fig. 1.1 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. Predicates such as `length/2`, `member/2` and `append/3` should be there – a programmer should not have to load them explicitly.[2] The public-domain set of libraries developed by the Prolog Commons Working Group provides a good start.[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 them. DCGs are not yet part of the Prolog ISO standard but are important enough to warrant inclusion in the Web Prolog language.
- As we are aiming for a profile of Prolog suitable for programming on the Web, predicates for working with the Web, e.g. `http_open/3`, and support for serialization as JSON 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.

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 we are lacking, and where we need to go.

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

- 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, we do not believe time is ripe to include them in Web Prolog. With time, as they mature, this may of course change.
- No (or limited use) of modules [TODO: Expand on this]

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`.

---

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

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

1.2.1 Web Prolog is inspired by Erlang

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

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, but was released as open source in 1998.[4] 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,[5] as well as for programming a wide range of soft real-time control problems [48].

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 three essential requirements:

1. The ability to execute a large number of actor processes concurrently;
2. the ability to use message-passing for inter-process communication, avoiding mutable shared memory and locking issues;
3. 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.

These are three 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 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.

---

[4] A good overview of the Erlang language, its design intent and the underlying philosophy, is given in the Ph.D. thesis of Joe Armstrong [3].

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

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 [4]. The following listings of Prolog code (to the left) and Erlang code (to the right) show some of the similarities as well as some of the differences between the two languages:

```
% append/3

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

```
% append/2

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

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 non-deterministically split a list up into two parts.

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, for example, typically play important roles in both kinds of languages.

Erlang was *losing* features that Prolog had (and still has), but *gained* powerful support for concurrent and distributed programming that Prolog did not have (and still does not have enough of). We are of course thinking of features such as 1, 2 and 3 in the above list of requirements. 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
    }).
```

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

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 of ISO Prolog.[6]

A function call such as `Pid = spawn(fun() -> echo_actor() end)` can be made in order to spawn an 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 goal to be passed in the first argument, a goal that when called will execute a procedure that determines how the actor will behave.

1.2.2 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 [47], depicted in Figure 1.2,[7] it is indeed possible to pinpoint exactly where Web Prolog fits in. We have enclosed the relevant square boxes in boxes with rounded corners.

This suggest 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.).

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 [47] 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

---

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

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

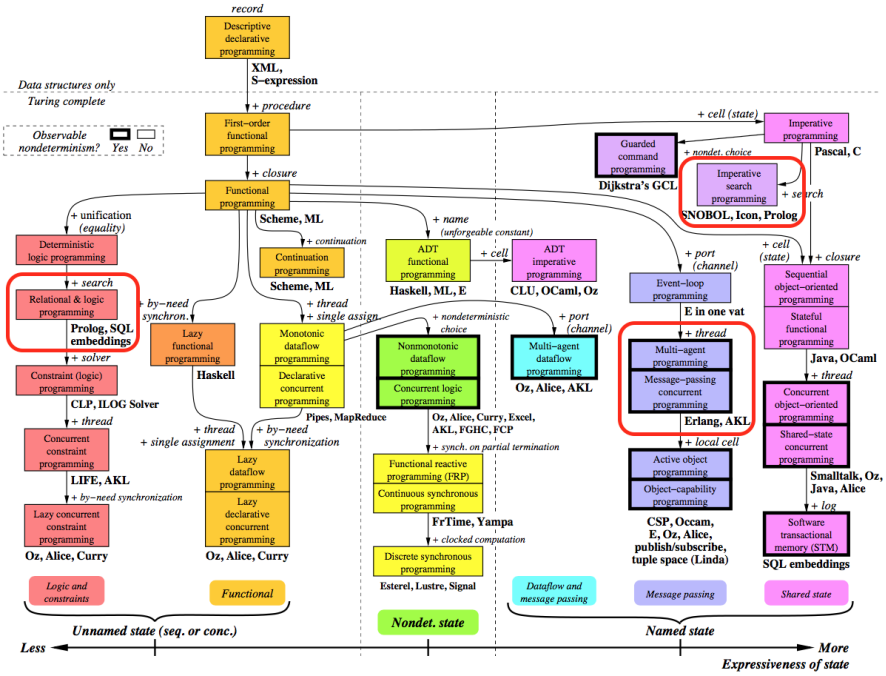

Fig. 1.2 A taxonomy of programming paradigms due to Peter van Roy.

programming language. In particular, it can be argued that the support for programming in the large has been greatly improved.

Still, 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, non-deterministic 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, in [2] Armstrong 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 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 should be clear by now 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.

Paradigms alone hardly capture all there is to a practical programming language. Languages may also be related by having a special purpose in common, or by constraints imposed upon them by a particular area of application. One such purpose, which comes with constraints pertaining to security, is the use of a language for programming the Web.

1.2.3 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. It appears that even back-end developers are more likely to use it than any other language.[8] 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

[8] http://stackoverflow.com/research/developer-survey-2016#most-popular-technologies-per-occupation

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 (Application Programming Interfaces) 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,[9] 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 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 1998 article "Scripting: higher level programming for the 21st Century,"[33] John Ousterhout 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

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

- secure.

As regards the viability of Web Prolog as a scripting language we note that while hardly a definition, Ousterhout characterizes scripting languages as follows in [33]:

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 we described above, it should now probably be described as general-purpose languages which also happens to be suitable for scripting. By contrast, some languages are not suitable for that purpose – C++ and Java comes to mind.

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.[10] 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:[11]

> 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 their language as glue:

> 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.[12]

On the Web – the biggest distributed programming system ever constructed – Erlang-style network-transparent message-passing concurrency indeed appears to be ideal for programming in the large, and indeed for the orchestration of both local and remote processes. But recall van Roy's assertion that a good language for large programs must support several paradigms, one paradigm for programming in the

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

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

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

small and another for programming in the large. With the addition of Erlang-style message-passing concurrency to Prolog's logic programming core supplemented by its imperative features, we end up with three paradigms. This cannot be a bad thing. It is likely that Erlang glue is of a different kind, with different properties, than Prolog glue. There is hope that the *combination* of Prolog and Erlang might result in an even stronger and more flexible glue of the kind required – 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 not *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.

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*,[13] with which the main JavaScript process is talking.

[13] https://en.wikipedia.org/wiki/Web_worker

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.

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.

1.3 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*

Inspired by Erlang's notion of an actor, the *Prolog actor* is the fundamental unit of computation in the Prolog Trinity ecosystem. The diagram in Figure 2.2 shows three actors, one of which has just been spawned, and one that has been opened up so that we can have a look inside (where on this level of abstraction they all look the same). 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. 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. Finally, there is a *spawn* operation that allows an actor to create other actors – child actors as it were.

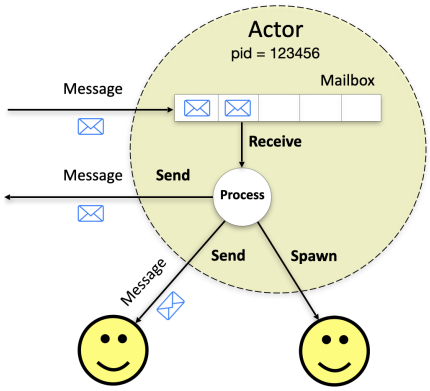

Fig. 1.3 The anatomy of a Prolog actor.

Corresponding to these operations there are built-in Web Prolog predicates such as `!/2`, `receive/1-2` and `spawn/2-3`. 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 proof-of-concept implementation.

Readers who are very experienced Prolog programmers and teachers might 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 and Erlang-style actor programming in the same system, and potentially 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.

1.3.1 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 Prolog tutorial, so how such queries are able to come up with results is not something we will explain. 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 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, waiting for a message to show up.

1.3.2 Two utility tools for programming in the shell

Having to copy and paste pids is not very convenient. Instead we can make use of a shell utility feature, borrowed from SWI-Prolog, which allows a variable binding resulting from the successful execution of a shell query to be reused in future shell queries if a dollar symbol is put in front of it, like so:

```
?- $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.

While being able to inspect the contents of the shell's mailbox during interactive programming is obviously important, using `receive/1` is not the most convenient way of doing it. After all, we cannot always be sure that there *are* any messages in the mailbox, and if it is empty `receive/1` will block and make normal interaction impossible. The call will just hang indefinitely, and the only way out may be to abort it using Control-C. Furthermore, we may want to see *all* messages in the mailbox which means that we need to run `receive/1` more than once, but we may not know how *many* messages there are, and then again run into the blocking problem. A more convenient tool here is the utility predicate `flush/0`. This predicate will show (and remove) all messages from the mailbox and will never block.

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.

1.3.3 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 creating 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 that is 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:

```
?- 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 exception, 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.

1.3.3.1 Monitoring

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. In our previous example, the shell and the echo server entered into precisely this relationship and thus formed a (very shallow!) hierarchy.

In a call such as `spawn(Goal,Pid,Options)` the optional third argument is a list of options used for the configuration of the actor. Two of these options – `monitor` and `link` – can be used to specify what should happen when the actor terminates, i.e what the parent might learn about the reason for its death, and how its children will be treated when it dies.

If the value of the `monitor` option is set to `true` the actor under construction is instructed to inform the parent about what eventually will become its fate. Let us look at a simple example:

```
?- spawn(append([a,b],[c,d],Zs), Pid, [
       monitor(true)
   ]).
Pid = 60367387.
?- flush.
Shell got down(60367387,true)
true.
?-
```

Here, the `monitor` option is 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. The idea is to allow the parent process to observe the child process and to detect if it has terminated for any reason. In this case, the `down` message was delived to its parent as soon as the goal terminated. Normally, an actor process will terminate and completely disappear when it has run out of things to do. It may be because the script succeeds, or fails, or throws an error. In this case, the atom `true` in the second argument of the `down` message tells us that it terminated in a way that can be considered normal.[14]

One more thing about this example deserves a comment. Note that although the goal `append([a,b],[c,d],Zs)` succeeds, the variable `Zs` is not bound. This is a consequence of the fact that the goal is *copied* before being run in the new process. (This is how it works in Erlang too.) Because of that, the only way for the parent process to get hold of the result of the call is to send it to the parent, like so:

```
?- self(Self),
   spawn((append([a,b],[c,d],Zs), Self ! Zs), Pid, [
       monitor(true)
   ]).
Self = 85234512,
Pid = 77129823.
```

[14] Erlang uses the atom `normal` to indicate this.

```
?- flush.
Shell got [a,b,c,d]
Shell got down(77129823,true)
true.
?-
```

1.3.3.2 Linking

If the value of the `link` option is `true` it means that when the actor terminates, any children that it might have are forced to terminate too. So in the following example, if the execution of `foo/0` spawns other actors (and the link only makes sense if it does), then should the `foo/0` call terminate, these actors will automatically be killed.

```
?- spawn(foo, Pid, [link(true)]).
Pid = 45092311.
?-
```

Note that it does *not* mean that the `foo/0` call must terminate if any child that it may have spawned terminates. The link is *uni-directional,* 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 bi-directional, and we shall discuss this difference further ahead.)

In the following example, because `link` is set to `false` our echo server will not necessarily terminate if the *shell* is terminated.

```
?- spawn(echo_actor, Pid, [link(false)]).
Pid = 66720916.
?-
```

In our proposal, the default value for `link` is `true`, as this is deemed to be what we usually want. In fact, and for a reason that will be explained later, in the context of distributed web programming it might be a good idea to *require* that the value of `link` is set to `true`.

1.3.3.3 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.
true.
?- 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.

1.3.3.4 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 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, 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. 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,'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, we were able to terminate the process. 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.

1.3.4 A closer look at receive/1-2

> I would argue that it is precisely the 'receive' construct in Erlang that makes Erlang such a joy to use.[15]
>
> *Richard O'Keefe*

[15] https://groups.google.com/forum/#!msg/erlang-programming/gjU-HCoq7dk/Mx_Af0iQ5P0J

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.

1.3.4.1 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, a receive call looks like this:

```
receive({
    Head1 -> Body1 ;
    Head2 -> Body1 ;
    ...
    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.

In its simplest form, the `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({
                continue(Msg) ->
                    format("Continuing with ~s~n", [Msg])
            })
    })
```

1.3.4.2 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 :-
    receive({
        foo ->
            process_foo,
            wait_bar
    }).
```

```
wait_bar :-
    receive({
        bar ->
            process_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.

1.3.4.3 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 big enough to warrant a different name for the operator.

1.3.4.4 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.

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

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

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.

1.3.4.5 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.

1.4 Erlang-style programming 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 kind of priority queue, a count server, an event-driven state machine, a universal stateful server supporting hot code swapping, actors playing ping-pong, a program solving goals in parallel, and an approach to building simple supervision hierarchies.

While most of the demonstrated actors have a behavior that is fully determined at the time of their creation, we also show how generic actors can be programmed (or reprogrammed) post creation-time.

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 behind a dedicated predicate API.

We have described Web Prolog as a language for distributed programming, but this chapter will only deal with local concurrency. Distributed programming will be dealt with in Chapter 3. For more exhaustive documentation of all built-in predicates available in Web Prolog, consult Appendix A.

1.4.1 A priority queue example

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 [16]. 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 toplevel 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.

1.4.2 An event-driven state machine

Web Prolog's receive primitive can be used to implement simple event-driven state machines in straightforward code. Patterns in receive clauses can be used to match against event messages, guards to encode conditions, and the bodies of receive clause to perform actions.

Figure 1.4 depicts a machine comprising two states and four transitions.

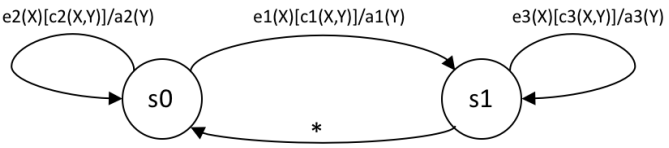

Fig. 1.4 A simple event-driven state machine

Using predicate names to encode states, here is how it can be implemented:

```
s0 :-
    receive({
       e1(X) if c1(X,Y) ->
           a1(Y),
```

```
s1 :-
    receive({
       e3(X) if c3(X,Y) ->
           a3(Y),
```

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

In state `s0` the machine is waiting for an event message of the form `e1(X)` or `e2(X)` to appear in the mailbox of the current process. If `e1(X)` is matched and the condition `c1(X,Y)` holds, the action `a1(Y)` will be called and the machine will transition to `s1`, but if `e2(X)` is matched and `c2(X,Y)` holds, then `a2(Y)` is called and the machine stays in state `s0`. And so on.

As far as we aware, event-driven state machines have not been used much in the Prolog world. Perhaps the reason is that primitives for sending and receiving messages did not appear in Prolog until (a few) platforms implemented the ISO Prolog Threads draft standard. As can be seen by an Erlang/OTP behavior such as `gen_statem`,[16] Erlang takes event-driven state machines very seriously, and perhaps it is time for Prolog to follow suit. We shall assume that it is, and in Chapter **??** present a proposal for how to introduce Web Prolog as a scripting language for StateChart XML, a W3C standard which provides an XML-based notation for event-driven state machines of a particularly sophisticated kind.

1.4.3 A stateful 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

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

call. Note how the state of the counter – the current count, that is – is kept in the argument of the predicate.

Note 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.
?-
```

Here, the `monitor` option is 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. Default is to not monitor.

Calling `self/1` determines the identity of the toplevel process – the process that just became the parent of the spawned actor:[17]

```
?- 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:

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

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

1.4.4 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 [16]:[18]

```
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:

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

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

1.4.5 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.

1.4.6 A universal stateful server with hot code swapping

Inspired by one of Joe Armstrong's lectures on Erlang,[19] this is how we may program a *generic* and *stateful* server in Web Prolog which can also handle *hot code swapping*:

```
server(Pred, State0) :-
    receive({
        rpc(From, Ref, Request) ->
            call(Pred, Request, State0, Response, State),
            From ! Ref-Response,
            server(Pred, State);
        upgrade(Pred1) ->
            server(Pred1, State0)
    }).
```

The first receive clause matches incoming `rpc` messages specifying a goal, performs the required computation, and returns the answer to the client that submitted the goal. The second clause is for upgrading the server.

As can be seen from the first receive clause, the generic server expects the definition of a predicate with four arguments to be present and callable from the server. In the case of our refrigerator simulation the expected predicate may be defined as follows in order to obtain the required specialisation of the generic server:

```
fridge(store(Food), FoodList, ok, [Food|FoodList]).
fridge(take(Food), FoodList, ok(Food), FoodListRest) :-
    select(Food, FoodList, FoodListRest), !.
fridge(take(_Food), FoodList, not_found, FoodList).
```

[19] http://youtu.be/0jsdXFUvQKE

Let us abstract from the message and make sure that the communication between client and server stays synchronous:

```
rpc_synch(To, Request, Response) :-
    self(Self),
    make_ref(Ref),
    To ! rpc(Self, Ref, Request),
    receive({
        Ref-Response -> true
    }).
```

Note the generation of a unique reference marker to be used to ensure that answers pair up with the questions to which they are answers. This code too is generic and follows a common idiom in Erlang that implements a synchronous operation on top of the asynchronous send and the blocking receive. The predicate `rpc_synch/3` waits for the response to come back before terminating. It inherits this blocking behavior from `receive/1`, and it is this behavior that makes the operation synchronous. (We will see more of this pattern later.)

Here is how we start a server process and then use `rpc_synch/3` to talk to it:

```
?- spawn(server(fridge, []), Pid).
Pid = 97216744.
?- rpc_synch($Pid, store(meat), Response).
Response = ok.
?- rpc_synch($Pid, take(meat), Response).
Response = ok(meat).
?-
```

Now, suppose we want to upgrade our server with a faster predicate for grabbing food from the fridge, perhaps one that uses an algorithm more efficient than the sequential search performed by `fridge/4`. Assuming a predicate `faster_fridge/4` is already loaded and present at the node we can make the upgrade without first taking down the server. This means that we can retain access to the state (and thus not risk losing any food in the process):

```
?- $Pid ! upgrade(faster_fridge).
true.
?-
```

Of course, the server can be reprogrammed in a much more radical way, and not only become faster, but also be given a totally different behavior.

1.4.7 Making promises, and keeping them

Remote procedure calls being synchronous means the caller is suspended until the computation terminates and we have to do an idle wait for the answer, although we

may have something better to do. An alternative approach may be to use a so called *promise*, an asynchronous variant of a remote procedure call. To implement this, we can split the client code above into two parts, and thus create two predicates, `promise/3` and `yield/2`:

```
promise(To, Request, Ref) :-
    self(Self),
    make_ref(Ref),
    To ! rpc(Self, Ref, Request).

yield(Ref, Response) :-
    receive({
        Ref-Response -> true
    }).
```

We can now separate the sending of a request from the receiving of the response, thus trading a somewhat messier code for a little bit of concurrency where the caller can perform the RPC, do something else and try to claim the computed value at a later time, when it may (or may not) be ready. Like so:

```
...
promise(To, store(meat), Ref),
... do something else here...
yield(Ref, Response),
...
```

With `promise/3` and `yield/2` defined, we can of course define `rpc_synch/3` as follows instead of as above:

```
rpc_synch(To, Request, Response) :-
    promise(To, Request, Ref),
    yield(Ref, Response).
```

Abstractions such as these can be compared to Erlang *behaviors*. They are not always easy to build, but once they are built they can be fairly easily instantiated and tailored to specific tasks.

1.4.8 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,[20] two processes are first created and then start sending messages to each other a specified number of times:

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

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

pong :-
    receive({
        finished ->
            format('Pong finished',[]);
        ping(Ping_Pid) ->
            format('Pong received ping',[]),
            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
?-
```

This is a concurrent program, but its execution is not done in parallel, but rather as is illustrated in Figure 1.5 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).

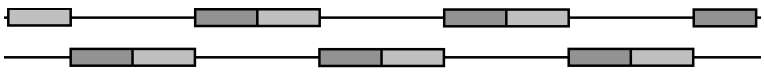

Fig. 1.5

In Appendix **??** 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 **??** also shows that Erlang is even faster.

1.4.9 Parallel execution of concurrent programs

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 `parallel/1` to work properly, we must require something about the input too, namely that goals in the list are independent, i.e. they must not communicate using shared variables, or by any other means.

To make it easier to understand what is going on, we begin by writing a predicate `parallel/2` that only takes *two* goals and spawns *two* actor processes that 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 the 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/2-3` 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 generalized 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({
        down(Pid, true) ->
            true ;
        down(_, false) ->
```

```
            tidy_up_all(Pids),
            !, fail ;
        down(_, error(E)) ->
            tidy_up_all(Pids),
            throw(E)
    }),
    receive({Pid-Goal -> true}).
```

In this version, all three predicates from the previous program have been modified. In `par_solve/2` each worker process 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(Pid, Reason)` arriving from the worker processes and act appropriately. If `Reason` is `true` nothing needs to be done, if `Reason` is `false` we call `fail` as that will make `parallel/1` fail, and if `Reason` is of the form `error(E)`, `E` is rethrown.

Note the use of anonymous variables in the first arguments of the patterns matching `false` and `error(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 `process/1` has a bit of tidying up to do, which will be performed by a call to `tidy_up_all/1`. For this it needs access to the complete list of pids for the worker actors which has been passed along since it was computed in the first call to `maplist/3` in the body of the clause defining `parallel/1`.

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),
    mailbox_rm(Pid).

mailbox_rm(Pid) :-
    receive({
        Msg if arg(1, Msg, Pid) ->
            mailbox_rm(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 with `Pid` in its first argument that may have reached the mailbox during the execution of `parallel/1` are removed.

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. This, of course, is as good as it gets. In terms of timelines, instead of seeing

Fig. 1.6 One actor performing three tasks sequentially.

we see this:

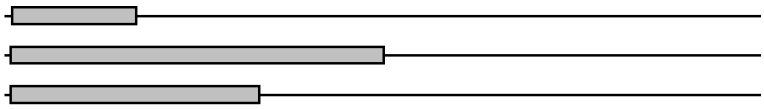

Fig. 1.7 Three actors performing three tasks in parallel.

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 we can catch:

```
?- _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, _)).
?-
```

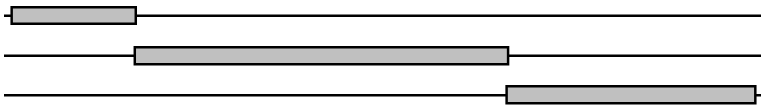

Fig. 1.8 Three actors performing three tasks concurrently, but not in parallel.

Consider the timeline in Figure 1.8. What if we have seen something like that instead? The processing would still be concurrent, but your computer might have just one core. A speedup can only be had if the computer on which the call is made is a multi-core machine.

Another 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 noticable difference, the predicate must be fed goals that run for a considerable time.

A predicate such as `parallel/1` should probably be available as a built-in in Web Prolog, and it is encouraging to see that it can be implemented as succinctly as this.

1.4.10 Creating supervision hierarchies

A *supervision hierarchy* refers to a structural pattern commonly used in concurrent and distributed systems, particularly in actor-based models like those found in Erlang. In such a system, actors are organized in a tree-like hierarchy where parent actors supervise their children. This structure is deemed crucial for fault tolerance and system resilience. Supervisors monitor their child actors for errors and decide on strategies (restart, stop, etc.) when errors occur. This ensures that the system can recover from errors locally without affecting the entire system's stability.

Here is code for a simple restarter procedure that takes a goal, a name to be given to the actor executing this goal, and an integer specifying the number of times a restart should be attempted before giving up:

```
restarter(Goal, Name, Count) :-
   spawn(restarter_loop(Goal, Name, Count), _, [
      monitor(true)
   ]).

restarter_loop(Goal, Name, Count0) :-
   spawn(Goal, Pid, [
      monitor(true)
   ]),
   register(Name, Pid),
   receive({
```

```
        down(Pid, true) ->
           true ;
        down(Pid, _Anything) ->
           (   Count0 == 0
           ->  true
           ;   Count is Count0 - 1,
               restarter_loop(Goal, Name, Count)
           )
      }).
```

Here is how it will work ([TODO: expand this!]):

```
?- restarter(echo_actor, echo_actor, 3).
true.
?- self(Self), echo_actor ! echo(Self, hello).
Self = 77129834.
?- flush.
Shell got echo(hello)
true.
?- whereis(echo_actor, Pid), exit(Pid, restart).
Pid = 94203114.
?- self(Self), echo_actor ! echo(Self, hello).
Self = 77129834.
?- flush.
Shell got echo(hello)
true.
```

1.5 Erlang-style programming beyond what Erlang can do

1.5.1 Getting answers through backtracking

In Chapter 1 we stated that, at least in theory, unexpected interactions between language features and possible impedance mismatches between Prolog's relational, non-deterministic 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 `?-mortal(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(( mortal(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.

1.5.2 A simple Prolog toplevel actor

In essence, a Prolog toplevel is a failure-driven loop running inside a tail-recursive loop. The failure-driven inner loop allows a client to ask for one solution at a time to

a goal, while the outer recursive loop allow 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.[21]

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

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

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

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

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

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

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

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

Note that the code for `simple_toplevel/1-2` does not say anything about what should happen if an error is thrown, which would for example be the case if the predicate called by the goal is not defined. If the spawned process is monitored, however, the error message will eventually reach the mailbox of the spawning process anyway, in the form of a `down` message. This means that the actor has terminated.

As we have seen, it is possible for a programmer to access and process different results of a non-deterministic computation from within a program. This is sometimes referred to as *encapsulated search*. But for encapsulated search to become a key feature of the ACTOR profile of Web Prolog, we need to provide something still more generic.

As it turns out, however, and as we shall demonstrate in Chapter 2, Web Prolog programmers do not need to define their own toplevel predicates, but can instead use a set of built-in predicates in order to create and control more powerful Prolog toplevels, which are actors adhering to a standardized and much improved protocol.

Chapter 2
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 **??** introduced the notion of a Prolog agent. In Chapter 1 we introduced 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 kind of agents, Prolog toplevels and Prolog nodes, will be introduced and their roles in the ecosystem explained.

The reason for referring to both actors and nodes as agents is to focus on their similarities. The similarity that makes use 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.

2.1 The concept of an agent

In Russell and Norvig [38] an agent is characterized as "anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors." It does not get more abstract than that, as it covers just about everything from a simple reactive agent such as a thermostat to a complex cognitive agent such as a human.

In this book, however, we shall be somewhat more concrete and care mostly about the concept of a *software agent*, i.e. an agent implemented in software and running

on computer hardware. The concept of software agent might be more appropriately defined as a software process capable of continious interaction with the world external to it. This definition of agenthood is admittedly very simple. It leaves out properties such as autonomy and goal-directedness, properties other researchers use in order to demarcate software agents from ordinary executing programs.[1] By our definition, any running program is an agent, as long as it is an *interactive* program.[2]

Software agents living in web-based environment form a category on its own. We shall refer to them as *web agents*. A web agent, in the context of the internet and web technology, typically refers to a program or script that performs automated tasks or interacts with web services on behalf of a user or another program. Some such agents are also known as *bots* or *web crawlers*. They can serve various purposes, such as web scraping, data collection, automated form submission, or even chatbots that interact with users on websites.

In the context of computer science and artificial intelligence, "spawning an agent" typically refers to the creation or instantiation of a new software agent from an existing one, resembling the parent-child relationship seen in biological reproduction. This process is akin to the notion of giving birth, where a "parent agent" generates a "child agent."

The parent agent is responsible for initiating and managing the child agent's execution. This can involve allocating resources, defining the child agent's initial state, and establishing communication channels between the parent and child agents. The child agent inherits certain characteristics or behaviors from its parent but may also possess its own unique attributes or functionalities.

This concept is frequently encountered in multi-agent systems, distributed computing environments, and parallel processing scenarios, where the dynamic creation of agents allows for increased flexibility and scalability in handling tasks and solving complex problems.

2.2 Prolog agents in a nutshell

A *Prolog agent* is an executing software process on the Prolog Web which is capable of continuous interaction with other Prolog agents in its environment using Web Prolog as a dedicated agent communication language (or an ACL, to use the commonly employed acronym). In this book, we deal primarily with Prolog agents that live on the Web, or in other words, that are *web agents*. Note that if taken as a definition it does not require that a Prolog agent is *written* in Web Prolog, only that it can *talk* Prolog.

---

[1] We agree with Russell and Norvig in [38] when they state that "[the] notion of an agent is meant to be a tool for analyzing systems, not an absolute characterization that divides the world into agents and non-agents," so we do not elaborate further on this.

[2] Thus, Turing machines are not agents, since they are not interactive.

As suggested by the taxonomy in Figure 2.1, Prolog agents come in different shapes and sizes, and as suggested by the diagram, they can be arranged into a simple taxonomy.

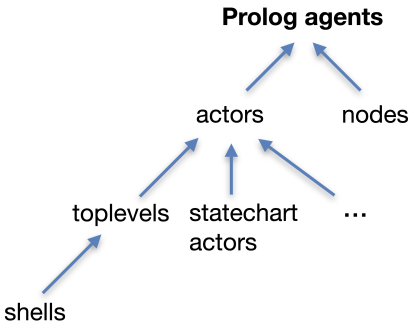

Fig. 2.1 A taxonomy of Prolog agents.

Prolog nodes (or just nodes) serve as the runtime systems of Web Prolog, and in the context of this book they will be regarded as agents. *Prolog actors* – very similar to what the Erlang community refer to as processes – are the loci of computation in the Prolog Trinity ecosystem. Prolog actors satisfy our definition of agenthood since they are processes capable of talking to other processes in their environment. As actors, they are all equipped with a mailbox from which messages received can be selected, and they can send messages to other agents. An actor can also spawn a child actor configured in all sorts of ways by means of options.

There are different *kinds* of actors such as *toplevels* and *statechart actors*, and there is a potential for other kinds of actors with other behaviors, some very specific, others generic. The ... in the diagram is a place holder for future such agents, and also covers actors such as echo actors and count actors. Chapter 1 looked at many examples of such actors in action.

An explanation of what we mean by a *statechart actor* will have to wait until Chapter **??**. Suffice it to say that it is a way to program an actor using a (partly) visual programming language called *statecharts*, invented by David Harel (who is featured among the other inventors in Chapter **??**). The relevancy to the Prolog Trinity ecosystem is that such actors can use Web Prolog as a data modelling and scripting language.

If toplevels and other actors are considered agents, it follows that they represent a rudimentary form of agency, characterized by minimal complexity in decision-making and autonomy. While these agents exhibit basic properties such as autonomy, interaction, and task-specific behavior, these characteristics alone may not suffice for what might be considered a fully autonomous agent. A more robust agent, such as a "soft robot" interacting with and adapting to its environment, would require additional cognitive capabilities, such as the ability to form desires and intentions, as outlined in the belief-desire-intention (BDI) framework.

Despite this, we remain satisfied with our simple agents, whether they function as nodes, toplevel actors, statechart actors, or other specialized entities. These agents have proven to be incredibly useful in their current form, particularly in web environments where their simplicity allows for ease of deployment and scalability. The utility of these agents is further enhanced by their ability to be composed into more complex systems, enabling the emergence of more sophisticated behaviors.

Moreover, it is important to recognize that "real" agents can be constructed from networks or societies of these simple agents. This modular approach enables the development of more advanced agents, such as those adhering to the BDI model, by building on the cooperative interactions of basic components.

2.3 More about Prolog actor agents

In the Erlang community, many actors, including actors with a very specific functionality such as echo actors and count actors, but also those that are generic such as compute servers, can be described in this way. Of, course, there are clients too, and other programs that cannot be described in this way.

Most actors comes with a communication protocol, i.e. a set of rules and conventions that govern how different actors (and other software processes) interact and communicate with each other.

2.3.1 An actor agent is equipped with a private Prolog database

So far, we have been a bit vague on where the code being executed came from. The only hint given was that it might be stored in the node's *shared database*, defining data and programs that can be used by any actor that lives there. We shall say more about such code further ahead in this chapter, but first we will introduce an area *inside* the actor where code might reside, as well as a way to make data and programs available to an actor.

An actor agent is equipped with a private Prolog database, where programs and data that may only be accessed and used by this particular actor is stored. All actors have such a database, but initially it may be empty. When it is not empty, its content should be contrasted with the programs and data that are shared among all actors that are present on a node, predicates that are loaded into the node's shared Prolog database.

When performing the spawn operation, the (soon-to-become) parent actor can choose to populate the private database of the child actor with new programs and data not present in the node's shared database.

The clauses in this database represent the agent's private beliefs and skills – in contrast to beliefs and skills that it shares with other actor agents running on the same node.

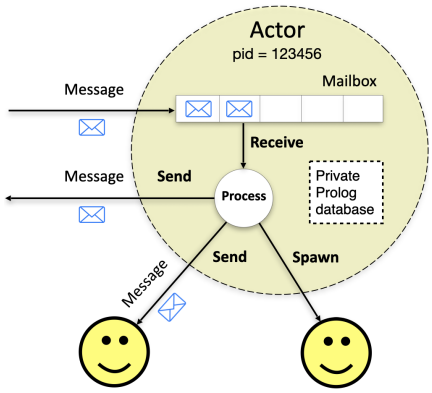

Fig. 2.2 The anatomy of a Prolog actor. Now complete with its private Prolog database.

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
               }).
       ")
   ]).
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` options 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.

It is possible to pass an arbitray number of the `load_*` options to `spawn/3`, and possibly more than one instance of each. To ensure that clauses end up in a well-defined order, they will all be converted into Prolog source text before finally being loaded into the database. The order of clauses and directives in the source text

is determined by the order of the `load_*` options in the list of options passed to `spawn/3`.

Note that the code that will eventually be executed by the call to `spawn/3` depends not only on what has been loaded into the private database of an actor, but also on the contents of the node's shared Prolog database. We shall return to this later in this chapter.

2.3.2 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 will only affect that actor's workspace, not actors running elsewhere, and not even actors running on the same node.

This means that problems caused by two or more processes trying to update the same database simultaneously cannot arise. It also means that since database updates performed by one process is not seen by other processes, assert and retract cannot be used for inter-process communication. Web Prolog adheres to the idea that processes should "communicate to share memory, rather than share memory to communicate".[3]

Therefore, although the following goal is permitted, it is quite meaningless since the clause `foo(a)` disappears as the goal has run and the spawned process has terminated.

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

The dynamic database provides another way to transfer from one state to the next. Here is an example:

```
?- spawn(count_actor, Pid, [
       load_text("
           :- dynamic cnt/1.

           cnt(0).

           count_actor :-
               receive({
                   count(From) ->
                       retract(cnt(Count0)),
                       Count is Count0 + 1,
                       From ! count(Count),
```

[3] A mantra attributed to CSP.

```
                        assert(cnt(Count)),
                        count_actor ;
                    stop ->
                        true
                }).
        ")
    ]).
Pid = 99801234.
?-
```

In this case, the use of the dynamic database is a very bad idea.

2.4 Prolog shells and other toplevel actors are agents

As we saw already in Chapter 1, if `self/1` is called in a terminal, its argument gets bound to an pid, and this pid points to a Prolog *shell*:

```
?- 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 adds 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/1` 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, the PTCP.

2.4.1 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

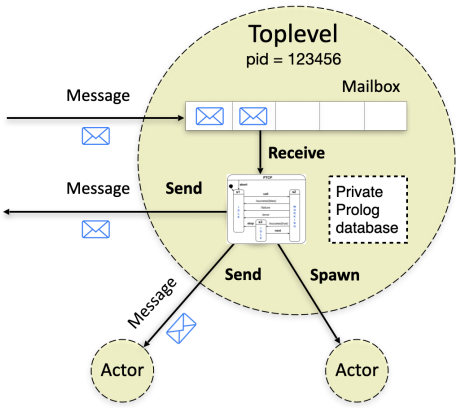

Fig. 2.3 The anatomy of a Prolog toplevel actor.

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 toplevel is *lazy* in the sense that new solutions to a query are only computed on demand. As we shall see, this is how the toplevel actor in Web Prolog 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, since 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.

2.4.2 The Prolog Toplevel Communication Protocol

Figure 2.4 depicts 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.

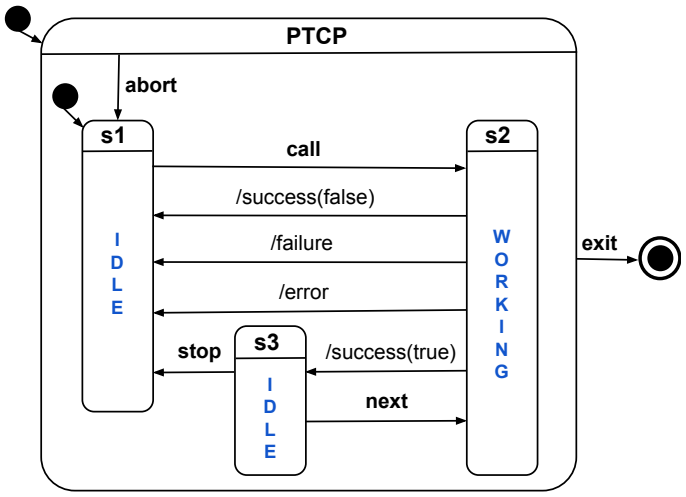

Fig. 2.4 Statechart specifying the PTCP for a successful conversation with a toplevel. The transitions are labeled with *message types*. Types in bold are sent from the client to the toplevel, whereas message types with a leading / goes in the opposite direction, from the toplevel to the client.

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

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

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

Such predicates are defined in terms of the more primitive `spawn/1-3`, `!/2` and `receive/1-2` predicates, but they also comes with a number of new options. Their uses will be demonstrated in the next section, and Appendix A contains an excerpt from the (draft) manual which covers most of the details of our proposal.

2.4.3 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/3` 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, [
       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/2-3`. However, `toplevel_spawn/3` offers two new options that can be used to specify the behavior

of toplevels that are spawned. The `session` option configures the toplevel to run a multi-query session rather than exit after just one query. It is true by default. The `target` option can be used to redirect the answer messages arriving from the toplevel to any actor of choice. By default, it is set to the pid of the parent. We say more about their use further ahead in this chapter.

2.4.3.1 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 a Prolog term with three arguments. The functor of the answer 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 was computed. The value `false` in the third argument of the term indicates that no more solutions exist.

After a brief visit to state **s2** for the execution of the goal, the PTCP is now back in state **s1**.

2.4.3.2 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.

2.4.3.3 Using the offset and limit options

When querying a relational database using SQL or an RDF dataset using SPARQL, the use of `OFFSET` and `LIMIT` serves to control the subset of results that are returned by a query.

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.

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

Calling `toplevel_next/1` produces three more solutions:

```
?- toplevel_next($Pid).
true.
?- flush.
Shell got success(74122981,[104,103,106],true)
true.
?-
```

However, `toplevel_next/2` accepts the `limit` option too:

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

2.4.3.4 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 usual send operator. However, a better idea can often be to use `output/1`, a built-in predicate that in its simplest form can be defined like so:

```
output(Message) :-
    self(Self),
    parent(Parent),
    Parent ! output(Self, Message).
```

Since the message is wrapped in a binary term with the pid of the toplevel in its first argument, the parent will always know if the message came from the right toplevel.

There is also `output/2`, which expect the option `target` to specify a target different from the parent.

(By the way, `output/1-2` can be used from any actor, not only from a toplevel.)

Now, let us demonstrate how I/O works:

```
?- 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).
?-
```

2.4.3.5 Aborting a non-terminating goal

The toplevel 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 non-terminating goal can be aborted by calling `toplevel_abort/1`:

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

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

2.4.3.6 Redirecting answers

The option `target` allows us to instruct a toplevel actor to send answer terms to a destination different from the parent. 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.
```

Its pid is used as the value of the target option:

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

The success term is printed, and calling `flush/0` shows that the mailbox belonging to the shell is empty:

```
?- toplevel_call($Pid, between(1,1000,N), [
       template(N),
       limit(5)
   ]).
true.
Received success(97919106,[1,2,3,4,5],true)
?- flush.
true.
?-
```

The `target` option is supported by all `toplevel_*` predicates, which allows us to direct the answer terms back to the shell. The only exception is `toplevel_stop/1`:

```
?- toplevel_stop($Pid).
Self = 34712309.
```

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

2.4.3.7 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:

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

Clearly, a Prolog toplevel is a kind of server (in the sense of Erlang – see Section 1.4.3). 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.

2.4.4 Toplevels and the message deferring mechanism

In the following example, a toplevel is spawned, and then `toplevel_next/1` is called. The protocol is obviously not in a state where it can react on the `next` message (see Figure 2.4). 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 (always?) 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 protocol 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 2.4). (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."[4] 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.

2.4.5 Reconstructing findall/3

If we did not already have a built-in predicate `findall/3` we could have implemented it like so:

```
findall(Template, Goal, Solutions) :-
    toplevel_spawn(Pid, [
        session(false)
    ]),
    toplevel_call(Pid, Goal, [
        template(Template),
        limit(none)
    ]),
    receive({
        success(Pid, Solutions, false) ->
            true ;
        failure(Pid) ->
            Solutions = [] ;
        error(Pid, Error) ->
            throw(Error)
    }).
```

It may not make much sense to do it like this, but at least it says something about the relation between `findall/3` and the `toplevel_*` predicates.

[4] https://en.wikipedia.org/wiki/Robustness_principle

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

Chapter 3
The Prolog Web

> Imagine the Web wrapped in Prolog, running on top of a distributed architecture comprised of a network of nodes supporting HTTP and WebSocket APIs, as well as 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 and allowing Web Prolog source code to flow in either direction, from the client to the node or from the node to the client. This is what **the Prolog Web** is all about.
>
> *The Prolog Web – the elevator pitch*

An important feature that Web Prolog has in common with Erlang is that concurrency is network transparent so that spawning and sending work also in a distributed setting. If we know the URI of a node and are authorized we can spawn an actor there, and if we know the pid of an actor process then we can send a message to it, even if it is running on another node. Indeed, all the mechanisms we have described – spawning, sending, linking, monitoring, registering, and so on – work transparently across node boundaries, making it easy to write distributed programs.

3.0.1 Actor talking to remote actor

For specifying the URI of the node on which we wish to create an actor process, the `spawn/3` predicate can be passed the option `node`. In the folllowing example, we use the `load_text` option to ship the predicate implementing our echo server to the node `http://n8.org`, invoke it there, monitor the process, and register it:

```
?- spawn(echo_actor, Pid, [
       node('http://n8.org'),
       load_text("
           echo_actor  :-
               receive({
                   echo(From, Msg) ->
                       From ! echo(Msg),
```

```
                        echo_actor
                }).
        "),
        monitor(true)
    ]),
    register(echo_actor, Pid).
Pid = 99231321@'http://n8.org'.
?-
```

Note that the pid is now a complex term and consists of a combination of an integer and a URI paired together by the *at sign* infix operator `@`.

Here is a short example of a conversation we can have with our remote echo server:

```
?- self(Self),
   echo_actor ! echo(Self, hello).
Self = 71773120.
?- flush.
Shell got echo(hello)
true.
?- whereis(echo_actor, Pid),
   exit(Pid, stop).
Pid = 99231321@'http://n8.org'.
?- flush.
Shell got down(99231321@'http://n8.org',stop)
true.
?-
```

3.0.2 Actors playing ping-pong

If the option `node('http://n2.org')` is passed to any of the calls to `spawn/3` in Section 1.4.8, the game of ping-pong will be played between two nodes:

```
ping_pong :-
    spawn(pong, Pong_Pid, [
        node('http://n2.org')
    ]),
    spawn(ping(3, Pong_Pid)).
```

Figure 3.1 illustrates this scenario.

But why do we get the output written by the "pinger" only, and not the output from the "ponger?" The reason is that a shell can only receive and render output that is written from actors that 1) live on the same node as the toplevel process to which the terminal is attached, and 2) are descendants of this toplevel process. (This

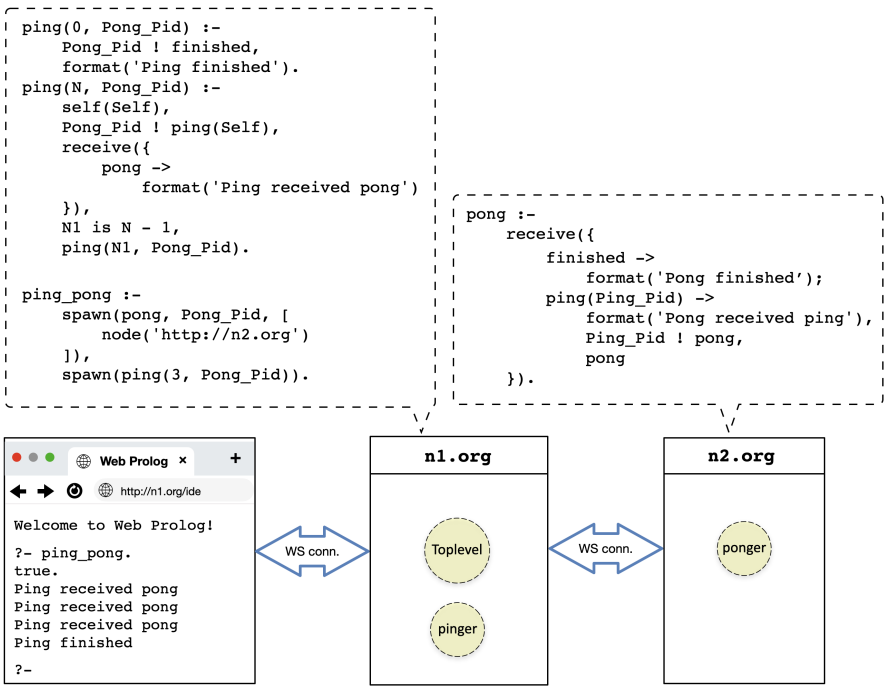

Fig. 3.1 A ping-pong scenario.

restriction may be lifted in the future.) In any case, our readers may rest assured that the actors really *do* play ping-pong.

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 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)).
```

In addition to the `load_predicates` option, `spawn/2-3` supports a number of other options which provide alternative ways to inject source code into the workspace of an actor.

3.0.3 Node-resident actor processes

In addition to node-resident source code, the owner of a node may install *node-resident actor processes*. We show an example below which uses `register/2` to give a running count actor a mnemonic name:

```
?- spawn(count_actor(0), Pid),
   register(counter, Pid).
```

Just like in Erlang, the registered name can be used instead of the pid when sending 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.
?-
```

Contrary to an actor injected and spawned by a client, a node-resident actor is accessible from any client to the node that knows its registered name. (This explains why 3 rather than 2 was received in the example – another client happened to increment the counter.)

As another example, here is all that is needed for a simple publish-subscribe service:

```
pubsub_service(Subscribers0) :-
    receive({
        publish(Msg) ->
            forall(member(Pid, Subscribers0), Pid ! msg(Msg)),
            pubsub_service(Subscribers0);
        subscribe(Pid) ->
            pubsub_service([Pid|Subscribers0]);
        unsubscribe(Pid) ->
            (   select(Pid, Subscribers0, Subscribers)
            ->  pubsub_service(Subscribers)
            ;   pubsub_service(Subscribers0)
            ).
    }).
```

We will assume the owner of a node has started the service by running the following goal:

```
?- spawn(pubsub_service([]), Pid),
   register(pubsub_service, Pid).
```

In the following example we subscribe to the service, and invoke a repeat-fail loop waiting for messages to arrive from it:

```
?- self(Self),
   pubsub_service ! 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 when someone with a connection to the same node published it to the service using the following kind of call:

```
?- pubsub_service ! publish(hello).
true.
?-
```

Node-resident actors are usually of a kind that we do not want unauthorized external clients to be able to create (or destroy). Doing this should be the privilege of the owner of the node or other authenticated users with the right authorities.

This is a simple pub-sub model designed for demonstration purposes, but it can be extended to meet various performance and scalability requirements. Possible improvements include increasing the fault tolerance by using a supervision tree to manage the actor process running `pubsub_service/1`, restart it if it crashes, and manage state recovery. Subscriptions might be stored in a more persistent store if needed, and we might want to optimize for large numbers of subscribers using more efficient data structures if scaling is needed.

3.0.4 The `node` option works for toplevels too!

It should come as no surprise that the `node` option can also be used to create a toplevel on a remote node. Suppose the node on which our shell runs have a shared database that contains the following clauses:

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

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

Also, suppose that we create a toplevel child on `http://n1.org` like so:

```
?- toplevel_spawn(Pid, [
       node('http://n1.org'),
       session(true),
       monitor(true),
       load_predicates([
           husband/2,
           wife/2
       ])
   ]).
Pid = 74122981@'http://n1.org'.
?-
```

Given the pid returned, `toplevel_call/3` can now be called with the goal `husband(Wife,Husband)`:

```
?- toplevel_call($Pid, husband(Wife,Husband)).
true.
?- receive({
       success($Pid, Data, false) ->
           true
   }).
Data = [husband(xantippa,socrates),husband(pythias,aristotle)]
?-
```

3.1 The sequential Prolog Web

On the concurrent Prolog Web, tasks are not only distributed, they are also executed during overlapping periods of time. This is a model that can be hard to control and where backtracking over nodes is performed in a way that may seem odd.

We will now present a model where tasks are distributed sequentially among different nodes in the network. Each node might perform its task in sequence, but overall, the system is still distributed because the tasks are spread across multiple nodes.

3.1.1 Non-deterministic remote procedure calls

In Web Prolog, `rpc/2-3` is a very high-level meta-predicate for making *non-deterministic remote procedure calls*. It allows a process running in a node N_1 to call and try to solve a query in the Prolog context of another node N_2, taking advantage of the data and programs being offered by N_2, just as if they were local to N_1. Such calls are synchronous, and this makes `rpc/2-3` remarkably easy to use for

the distribution of programs over two or more nodes. It is a predicate for distributed programming that trades concurrency for ease of programming.

A Web Prolog client processes queries at a remote node by making an `rpc/2-3` call with the first argument a URI pointing to the node, and the second argument a goal to be run over the predicates offered by it. The third argument expects a list of options. Here is an example of how `rpc/2-3` can be used:

```
?- rpc('http://n1.org', mortal(Who), [
       load_list([(mortal(Who) :- human(Who))])
   ]).
Who = plato ;
Who = aristotle.
?-
```

Note that solutions are given on demand and in the usual one-solution-at-a-time fashion, as bindings of variables that occur in the goal.

There are eight billion humans on Earth, so how would `rpc/3` cope if they were all represented by `human/1`?

```
?- rpc('http://n1.org', mortal(Who), [
       load_list([(mortal(Who) :- human(Who))]),
       limit(1000)
   ]).
```

The value `1000` of the option `limit` told `rpc/3` to limit the number of solutions it should compute during one network roundtrip to one thousand. So in this case it had to make eight million roundtrips.

A notable property of `rpc/2-3` is that it retains the logical purity of the goal that it calls, so that if the goal is pure, then the entire call is pure.[1] Further ahead we shall argue that it makes the notion of a *pure* Prolog Web a coherent proposition.

3.1.2 Implementing rpc/2-3 on top of a toplevel actor

Below, we show an implementation of `rpc/2-3` which is built on top of a toplevel spawned on a remote node and a local loop that waits for answers arriving from it:

```
rpc(URI, Goal) :-
    rpc(URI, Goal, []).

rpc(URI, Goal, Options) :-
    toplevel_spawn(Pid, [
        node(URI),
        session(false)
```

[1] This is a property which it shares with the `call/N` family of Prolog predicates.

```
        | Options
    ]),
    toplevel_call(Pid, Goal, Options),
    wait_answer(Pid, Goal).

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)
    }).
```

When spawning the toplevel we pass `session(false)` in order to make sure that the process terminates when the first query being sent to it has run to completion. Note also that we do not monitor the spawned process. Thus, when the query has run to completion, no `down` message is going to come along and report this, and so the process running the `wait_answer/2` predicate does not have to bother looking for this message in its mailbox. It can safely terminate on failure, error or when having visited the last solution in the last success message received from the node.

The first clause of the implementation shows clearly how the inheritance of options from `toplevel_spawn/2` and `toplevel_call/3` to `rpc/2-3` works. Note that the value of an option passed explicitly to `toplevel_spawn/2` takes precedence over the value of the same option if it's given in the third argument of `rpc/3`. Thus, the only options that can have an effect in a call to `rpc/3` are the `timeout` and the `load_*` options.

The most interesting parts of the implementation are the use of the disjunction in the body of the first receive clause and the use of `member/2` in the first and second clauses. They are responsible for turning the deterministic calls made by `toplevel_call/3` and `toplevel_next/1` into the non-deterministic behavior we want `rpc/2-3` to have.

The third receive clause is triggered by the appearance in the mailbox of a `failure` answer term, and simply causes `rpc/2-3` to fail, and if triggered by an `error` message the fourth clause rethrows the error.

Note that since the client actor and the remote toplevel do not execute different tasks during overlapping periods of time, no real parallelism is taking place here. Also, the communication is synchronous. Thus, `rpc/2-3` can only contribute to a partition of the Prolog Web that is synchronous and sequential.

3.1.3 Template packing and output elimination

While `rpc/2-3` does not support the `template` option, this option is useful in the *implementation* of `rpc/2-3` as it can reduce the size of the payload sent by `toplevel_call/3` over the network by almost half. The idea is to form a template by wrapping the goal variables in a term. To accomplish this, the ISO predicate `term_variables/2` can be used in combination with the so-called *univ* operator (`=..`). We refer to the technique as *template packing* and demonstrate it in the code below:

```
rpc(URI, Goal, Options) :-
    toplevel_spawn(Pid, [
         node(URI),
         session(false)
       | Options
    ]),
    term_variables(Goal, Vars),
    Template =.. [v|Vars],
    toplevel_call(Pid, Goal, [
          template(Template)
        | Options
    ]),
    wait_answer(Pid, Template).

wait_answer(Pid, Template) :-
    receive({
        success(Pid, Slice, true) ->
            (   member(Template, Slice)
            ;   toplevel_next(Pid),
                wait_answer(Pid, Template)
            ) ;
        success(Pid, Slice, false) ->
            member(Template, Slice) ;
        failure(Pid) -> !, fail ;
        error(Pid, Error) ->
            throw(Error) ;
        Any if arg(1, Any, Pid) ->
            wait_answer(Pid, Template)
    }).
```

Note that we now need to pass the template rather than the goal in the call to `wait_answer/2`, which on backtracking will be repeatedly unified with elements of the slices of solutions arriving from the remote toplevel process. Sharing of variables between the goal and the template takes care of the instantiation of the goal.

The last clause in the call to `receive/1` catches any other ouput that might arrive from the remote toplevel, such as messages of the form `output(Pid,Data)` or `prompt(Pid,Data)`. This is what we refer to as *output elimination*.

References

1. Alferes, J.J., Damasio, C.V., Pereira, L.M.: Semantic web logic programming tools. In: International Workshop on Principles and Practice of Semantic Web Reasoning (PPSWRâĂŹ03, pp. 16–32 (2003)
2. Armstrong, J.: Concurrency oriented programming in erlang. Invited talk, FFG (2003)
3. Armstrong, J.: Making reliable distributed systems in the presence of software errors. Phd thesis, Royal Institute of Technology, Stockholm (2003)
4. Armstrong, J., R. Virding, S., C. Williams, M.: Use of prolog for developing a new programming language (1995)
5. 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.: State chart xml (scxml) state machine notation for control abstraction. w3c recommendation (2015)
6. Berners-Lee, T., Hendler, J., Lassila, O.: The semantic web. Scientific American **284**(5), 34–43 (2001). URL http://www.sciam.com/article.cfm?articleID=00048144-10D2-1C70-84A9809EC588EF21
7. Brusk, J., Lager, T.: Developing natural language enabled games in SCXML. Journal of Advanced Computational Intelligence and Intelligent Informatics (JACIII) **12**(2), 156–163 (2008). DOI 10.20965/jaciii.2008.p0156. URL https://doi.org/10.20965/jaciii.2008.p0156
8. Byrd, L.: Understanding the control flow of Prolog programs. In: S. Tarnlund (ed.) Logic Programming Workshop, pp. 127–138. Debrecen, Hungary (1980)
9. Calegari, R., Denti, E., Mariani, S., Omicini, A.: Logic programming as a service. Theory and Practice of Logic Programming **18**(5-6), 846âĂŞ873 (2018). DOI 10.1017/S1471068418000364
10. Cicirelli, F., Furfaro, A., Giordano, A., Nigro, L.: 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 (2009). DOI 10.7148/2009-0233-0239. URL https://doi.org/10.7148/2009-0233-0239
11. Clark, K., J. Robinson, P., Hagen, R.: Multi-threading and message communication in qu-prolog **1**, 283–301 (2001)
12. Deliyanni, A., Kowalski, R.A.: Logic and semantic networks. Communications of the ACM **22**(3), 184–192 (1979)
13. Graham, P.: Hackers & painters: big ideas from the computer age. " O'Reilly Media, Inc." (2004)
14. Hachey, G., Gasevic, D.: Semantic web user interfaces: A systematic mapping study and review (2012)
15. Harel, D.: Statecharts: A visual formalism for complex systems. Sci. Comput. Program. **8**(3), 231–274 (1987). DOI 10.1016/0167-6423(87)90035-9. URL https://doi.org/10.1016/0167-6423(87)90035-9

16. Hebert, F.: Learn You Some Erlang for Great Good!: A Beginner's Guide. No Starch Press, San Francisco, CA, USA (2013)
17. Hendler, J.: Agents and the semantic web. IEEE Intelligent systems **16**(2), 30–37 (2001)
18. Hendler, J., van Harmelen, F.: The semantic web: Webizing knowledge representation. In: F. van Harmelen, V. Lifschitz, B. Porter (eds.) Handbook of Knowledge Representation, *Foundations of Artificial Intelligence*, vol. 3, pp. 821 – 839. Elsevier (2008). DOI https://doi.org/10.1016/S1574-6526(07)03021-0. URL http://www.sciencedirect.com/science/article/pii/S1574652607030210
19. Hoare, C.: Hints on programming language design. Tech. rep., STANFORD UNIV CA DEPT OF COMPUTER SCIENCE (1973)
20. Horrocks, I.: Constructing the user interface with statecharts. Addison-Wesley Longman Publishing Co., Inc. (1999)
21. Horrocks, I., Parsia, B., Patel-Schneider, P., Hendler, J.: Semantic Web Architecture: Stack or Two Towers?, pp. 37–41. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
22. Junger, D.: Client-side state-based control for multimodal user interfaces. Master's thesis, University of Gothenburg (2014)
23. Junger, D., Lager, T., Roxendal, J.: Scxml for building conversational agents in the dialog web lab (2012)
24. Kifer, M., de Bruijn, J., Boley, H., Fensel, D.: A Realistic Architecture for the Semantic Web, pp. 17–29. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
25. Kowalski, R.A.: Algorithm = logic + control. Commun. ACM **22**(7), 424–436 (1979). DOI 10.1145/359131.359136. URL https://doi.org/10.1145/359131.359136
26. Lager, T.: Intro to web prolog for erlangers. In: Proceeding of the 18th ACM SIGPLAN International Workshop on Erlang, pp. 18–29. ACM (2019). DOI 10.1145/3331542.3342569
27. Lager, T., Myrendal, J.: Exploring the web of coined catchy phrases. In: Proceedings of WWW2012: Web Science Track (2012)
28. Lager, T., Wielemaker, J.: Pengines: Web logic programming made easy. Theory and Practice of Logic Programming **14**(4-5), 539–552 (2014). DOI 10.1017/S1471068414000192. URL http://dx.doi.org/10.1017/S1471068414000192
29. Loke, S.W.: Declarative programming of integrated peer-to-peer and web based systems: the case of prolog. Journal of Systems and Software **79**(4), 523–536 (2006). URL http://dx.doi.org/10.1016/j.jss.2005.04.005
30. Loke, S.W., Davison, A.: Secure prolog-based mobile code. Theory Pract. Log. Program. **1**(3), 321–357 (2001). DOI 10.1017/S1471068401001211. URL http://dx.doi.org/10.1017/S1471068401001211
31. Lombardi, A.: WebSocket: Lightweight Client-server Communications. O'Reilly Media, Inc. (2015)
32. Merritt, D.: Building Expert Systems in Prolog. Springer-Verlag (1989)
33. Ousterhout, J.K.: Scripting: higher level programming for the 21st century. Computer **31**(3), 23–30 (1998)
34. Piancastelli, G., Omicini, A.: A Multi-theory Logic Language for the World Wide Web, pp. 769–773. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
35. Poole, D.: Representing diagnosis knowledge. Ann. Math. Artif. Intell. **11**(1-4), 33–50 (1994). DOI 10.1007/BF01530736. URL https://doi.org/10.1007/BF01530736
36. Radomski, S., Schnelle-Walka, D., Radeck-Arneth, S.: A prolog datamodel for state chart xml. In: Proceedings of the SIGDIAL 2013 Conference, pp. 127–131. Association for Computational Linguistics (2013). URL http://aclweb.org/anthology/W13-4019
37. van Roy, P., Haridi, S.: Concepts, Techniques, and Models of Computer Programming. MIT Press (2004)
38. Russell, S.J., Norvig, P.: Artificial intelligence: a modern approach. Malaysia; Pearson Education Limited, (2016)
39. Samek, M.: Practical statecharts in C/C++: Quantum programming for embedded systems. CRC Press (2002)

40. Schmid, U., Mandl, S., Gust, H., Kitzelmann, E., Helmert, M., Buschmeier, H., Yaghoubzadeh, R., Pietsch, C., Kopp, S., Hertzberg, J., Sprickerhof, J., Wiemann, T.: What language do you use to create your ai programs and why? Künstliche Intelligenz **26**(1), 99–106 (2012). DOI 10.1007/s13218-011-0158-z. URL https://doi.org/10.1007/s13218-011-0158-z
41. Schrijvers, T., Demoen, B.: Uniting the Prolog Community, pp. 7–8. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
42. Shieber, S.M.: An introduction to unification-based approaches to grammar. Microtome Publishing (2003)
43. Skantze, G., Moubayed, S.A.: 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 (2012). DOI 10.1145/2388676.2388698. URL https://doi.org/10.1145/2388676.2388698
44. Smith, R.W., Biermann, A.W., Hipp, D.R.: An architecture for voice dialog systems based on prolog-style theorem proving. Computational Linguistics **21**(3), 281–318 (1995). URL http://www.aclweb.org/anthology/J95-3001
45. Sterling, L., Shapiro, E.: The Art of Prolog - Advanced Programming Techniques, 2nd Ed. MIT Press (1994)
46. Svensson, H., Fredlund, L.Å., Earle, C.B.: A unified semantics for future erlang. In: S.L. Fritchie, K.F. Sagonas (eds.) Erlang Workshop, pp. 23–32. ACM (2010). URL http://dl.acm.org/citation.cfm?id=1863509
47. Van Roy, P.: Programming paradigms for dummies: What every programmer should know. New computational paradigms for computer music **104** (2009)
48. Virding, R., Wikström, C., Williams, M., Armstrong, J.: Concurrent programming in ERLANG (2nd ed.). Prentice Hall International (UK) Ltd., GBR (1996)
49. Vlassis, N.: A concise introduction to multiagent systems and distributed artificial intelligence. Synthesis Lectures on Artificial Intelligence and Machine Learning **1**(1), 1–71 (2007)
50. Wielemaker, J., Beek, W., Hildebrand, M., van Ossenbruggen, J.: Cliopatria: A swi-prolog infrastructure for the semantic web. Semantic Web **7**(5), 529–541 (2016)
51. Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: SWI-Prolog. Theory and Practice of Logic Programming **12**(1-2), 67–96 (2012)

Appendix A
Excerpt from the draft manual

A.1 Predicates for programming with actors

Predicate: `self/1` ACTOR

```
self(-Pid) is det.
```

Binds `Pid` to the process identifier of the calling process.

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 on the node pointed to by the URI. Default is `localhost`.
- `monitor(+Boolean)`
  If `true`, sends a `down` message to the parent process when the spawned process terminates. Default is `false`.
- `link(+Boolean)`
  If `true`, terminates all child processes (if any) upon termination of the spawned process. Default is `true`.
- `timeout(+IntegerOrFloat)`
  Terminates the spawned process (or the process of spawning a process) after `IntegerOrFloat` seconds.
- `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)`
  Loads a list of Web Prolog clauses into the actor's private Prolog database before calling `Goal`.

- `load_uri(+URI)`
  Loads the clauses specified by the Web Prolog source text at `URI` into the actor's private Prolog database before calling `Goal`. `Goal`.
- `load_predicates(+ListOfPredicateIndicators)`
  Loads the local predicates denoted by `ListOfPredicateIndicators` into the actor's private Prolog database before calling `Goal`.
- `type(+Atom)`
  Indicates the type of the source to be injected into the process. Default is `'web-prolog'`. Note that some `load_*` options may not be compatible with other values of this option.

Predicate: `monitor/1` ACTOR

```
monitor(+Pid) is det.
```

Begin monitoring the process `Pid`.

Predicate: `demonitor/1` ACTOR

```
demonitor(+Pid) is det.
```

Stop monitoring the process `Pid`.

Predicate: `register/2` ACTOR

```
register(+Name, +Pid) is det.
```

Register a process under a name, where `Name` is an atom and `Pid` identifies the actor process. The association between the name and the pid will be removed when the process terminates.

Predicate: `whereis/2` ACTOR

```
whereis(+Name, ?Pid) is det.
```

Locate the process associated with the name. Returns `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: `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 as `Pid` with `Reason`.

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` must have the form `Pid@Node` or `Name@Node`. `Pid@localhost` and `Name@localhost` refer to actors running on the current node, and can often be abbreviated to `Pid` or `Name`, respectively. Valid options for `send/3` are:

- `delay(+IntegerOrFloat)`
  Delays the sending with 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 of the message to happen.

Predicate: `cancel/1` ACTOR

```
cancel(+ID) is det.
```

Tries to cancel the sending of *all* delayed messages with the specified ID. This cannot be guaranteed to succeed since a message may already have been sent by the time the call is made.

Predicate: `raise/1` ACTOR

```
raise(+Message) is det.
```

Sends `Message` to the mailbox of the current process. Bootstrapped as

```
raise(Message) :-
    self(Pid),
    Pid ! Message.
```

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, just like any other actor, may use `!/2` to send any term to any process to which it has a pid.

Predicate: `input/2` 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 term (i.e. even a compound 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 (the one that has been waiting longest) 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 the first message is not accepted, the second one will be tried, then the third, and so on. If none of the messages in the mailbox is accepted, the process will wait for new messages, checking them one at a time in the order they arrive. Messages in the mailbox that are not accepted are *deferred*, i.e. left in the mailbox without any change in their contents or order. Valid options:

- `timeout(+IntegerOrFloat)`
  If nothing appears in the current mailbox within `IntegerOrFloat` 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 and binds `Pid` to its pid. With just two exceptions, all options that can be passed to `toplevel_spawn/2` are inherited from `spawn/3`. The only new options are `session` and `target`.

- `session(+Boolean)`
  If set to `false`, the toplevel actor will terminate after having run a goal to completion. If `true`, further interaction is expected. Defaults to `false`.
- `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 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)`
  By default, `toplevel_call/2-3` requests that *all* solutions to `Goal` be computed and returned as a list of solutions embedded in an answer term of type `success`. By passing the limit option, the length of this list can be restricted to `Integer`.
- `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` will not be bound. Instead, solutions and other kinds of output will be returned in the form of answer messages delivered to the mailbox of the process that called `toplevel_spawn/2-3`.

- `success(Pid, Data, More)`
  `Pid` refers to the toplevel process that succeeded in solving the goal. `Data` is a list holding instantiations of `Template`. `More` is either `true` or `false`, indicating whether or not we can expect the toplevel to be able to return more solutions, would we call `toplevel_next/1-2`.
- `failure(Pid)`
  `Pid` is the pid of the toplevel process that failed for lack of (more) solutions.

- `error(Pid, Data)`
  `Pid` is the pid of the toplevel throwing the error. `Data` is the error term.

Note that nothing stops a toplevel from sending messages of a form different from the above to the target.

Predicate: `toplevel_next/1-2` ACTOR

```
toplevel_next(+Pid) is det.
toplevel_next(+Pid, +Options) is det
```

Asks toplevel `Pid` for more solutions to `Goal`. Valid options:

- `limit(+Integer)`
  By default, the value of the `limit` option is the same as for `toplevel_call/2-3`.
- `target(+Pid)`
  Send the answer term to `Pid`. Default is the value of `target` when passed as an option to `toplevel_call/3`.

The messages delivered to the mailbox of the target 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 any goal it currently runs.

Predicate: `toplevel_exit/1-2` ACTOR

```
toplevel_exit(+Reason) is det.
toplevel_exit(+Pid, +Reason) is det.
```

Same as `exit/1` and `exit/2`.

A.3 Built-in Predicates for RPC

Predicate: `rpc/2-3` ISOBASE

```
rpc(+URI, +Goal) is nondet.
rpc(+URI, +Goal, +Options) is nondet.
```

Semantically equivalent to the sequence below, except that the goal is executed in (and in the Prolog context of) the node referred to by `URI`, rather than locally.

```
copy_term(Goal, Copy),
call(Copy),      % executed on node at URI
Goal = Copy.
```

The following options are valid:

- `limit(+Integer)` ISOBASE
  By default, `rpc/2-3` will only make one trip to the remote node at `URI` in which it will (try to) compute all solutions to `Goal` in order to cache them at the client. A goal with n solutions and `limit` set to 1 would require n roundtrips if we wanted to see them all. With `limit` set to i, the same goal would only require $ceiling(n/i)$ roundtrips.
- `timeout(+IntegerOrFloat)` ISOBASE
  Terminates the spawned process (or the process of spawning a process) after `IntegerOrFloat` seconds.
- `load_text(+AtomOrString)` ISOTOPE
  Loads the clauses specified by a Web Prolog source text into the underlying actor's private Prolog database before calling `Goal`.
- `load_list(+ListOfClauses)` ISOTOPE
  Loads a list of Web Prolog clauses into the underlying actor's private Prolog database before calling `Goal`.
- `load_uri(+URI)` ISOTOPE
  Loads the clauses specified by the Web Prolog source text at `URI` into the underlying actor's private Prolog database before calling `Goal`.
- `load_predicates(+ListOfPredicateIndicators)` ISOTOPE
  Loads the local predicates denoted by `ListOfPredicateIndicators` into the underlying actor's private Prolog database before calling `Goal`.
- `monitor(+Boolean)` ACTOR
  Default is `false`, i.e. to not monitor. The node at `URI` must be another ACTOR node.
- `protocol(+Atom)` ACTOR
  If `Atom` is `http` (default), the HTTP protocol will be used as transport, and if `Atom` is `ws`, a WebSocket connection will be used.
- `pid(-Pid)` ACTOR
  The `pid` option is passed with a free variable `Pid` which will be bound to the pid of the remote toplevel when the call returns. Using the `pid` option breaks the abstraction for remote procedure calling, so it should be used with care. Note that if `transport` is `http`, `Pid` will be bound to `anonymous`. The node at `URI` must be another ACTOR node.

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`. This is a type of RPC which does not suspend the caller until the result is computed. Instead, a reference `Ref` is returned, which can later be used by `yield/2-3` to collect the answer. The reference can be viewed as a promise to deliver the answer. Valid options are:

- `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)`
  By default, `promise/3-4` requests that *all* solutions to `Goal` be computed and returned as a list of solutions embedded in an answer term of type `success`. By passing the limit option, the length of this list can be restricted to `Integer`.

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`. If the answer is available, it is returned immediately. Otherwise, the calling process is suspended until the answer arrives from the node that was called. Note that this predicate must be called by the same process from which the previous call to `promise/3-4` was made, otherwise it will not return. Valid options:

- `timeout(+IntegerOrFloat)`
  If nothing appears in the current mailbox within `IntegerOrFloat` seconds, the predicate succeeds anyway. Default is no timeout.
- `on_timeout(+Goal)`
  If the timeout occurs, `Goal` is called.

A.4 The stateless HTTP API

In our proposal for an HTTP query API, the URI in a GET request for one or more solutions to a query has the following form:

```
BaseURI/call?goal=G&template=T&offset=O&limit=L&format=F
```

The URI denotes a resource in the form of a (possibly only partial) answer to the goal `G` as given by the node `BaseURI`. The template `T` works as in `findall/3` and the semantics of the `offset` and `limit` parameters are borrowed from SQL and SPARQL, As in these languages they expect integer values, where `offset` defaults to `0` and `limit` to `infinite`. A client may also use a parameter `load_text` in order to send along source code to complement the goal. Responses are returned as Prolog terms or as Prolog variable bindings encoded in JSON.

A response contains a success answer, a failure answer, or an error answer. A success answer contains a list of solutions of the form `T` to `G`, starting at offset `O` and having a length of at most `L`. In addition, an indication whether more solutions may exist is given. By default, answers are rendered as JSON, where the slice of solutions is represented as a list of pairs of the form `{<var>:<value>,..,<var>:<value>}`. A failure answer indicates that no (more) solutions exists, and an error answer signals an error and carries an error message.

[TODO: Needs more work!]

A.5 The stateful WebSocket API

[TODO: Needs work!]

Appendix B
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?

B.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 implementor, one obvious advantage with this approach is that most of the necessary 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.[1] We are aiming for an almost complete ISOBASE node, as well as 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 spurious 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 also implement specifications for how we believe predicates such as `spawn/2-3`, `!/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.

B.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).

node(Port) :-
    http_server(http_dispatch, [port(Port)]).
```

[1] http://logtalk.org/plstd/threads.pdf

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 non-deterministic, so on backtracking it will do it again. We borrow an example of its use from the SWI-Prolog manual:[2]

```
?- 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.[3] 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)
```

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

[3] https://www.swi-prolog.org/pldoc/doc_for?object=offset/2

```
    ->  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.

B.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,[4] 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).
```

[4] 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 non-deterministic behavior we want `rpc/2-3` to show.

Let us test our implementation by running an example from Chapter 3, 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.

B.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.
?-
```

It is only when the `limit` option must be employed, so that more than one network roundtrip has to be made, 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.[5]

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.

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.[6]

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,

[5] Credits for this idea goes to Jan Wielemaker. The implementation is our's.

[6] 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),
                    toplevel_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.

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 **??**, 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.

B.1.4 Implementing the Erlang-style concurrency predicates

This section implement specifications for how we believe predicates such as `spawn/2-3`, `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 have found 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.[7]

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`

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.

Here is a first sketch of an implementation of `spawn/2-3`:

---

[7] https://logtalk.org/plstd/threads.pdf

```
:- 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, linked)),
        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)).
```

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/2-3` calls two predicates – `exit/2` and `!/2` – that must be implemented. In addition, `exit/1` must be implemented, and this can be done as followoms:

```
:- 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 predicates output/1-2, input/2-3 and respond/2 are implemented on top of the !/2 primitive. Their purpose is to simulate I/O.

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).
```

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
    ).
```

B.1.5 Implementing the first-class Prolog toplevel

In addition to the Erlang-style actors, the toplevel behavior, controlled by predicates such as `toplevel_spawn/1-2` and friends, must also be implemented. We refer the reader back to Chapter 2 for how this should work and for some hints for how it can 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(session(Session), Options, false),
    option(target(Target), Options, Self),
    spawn(state_1(Pid, Target, Exit), Pid, Options).
```

Note that options passed to `toplevel_spawn/2` will be passed on to `spawn/3` as well.

The most important part of the implementation of the PTCP protocol are the three states s1, s2 and s3 , depicted in the diagram in Figure B.1:

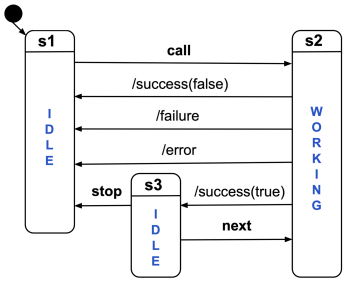

Fig. B.1 The three inner states of the PTCP protocol.

```
state_1(Pid, Target0, Session) :-
```

```
    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
            )
        }),
    (   Session == false
    ->  true
    ;   state_1(Pid, Target0, Session)
    ).
```

In `state_2` the real work is being done. The predicate `answer/5`, defined earlier in this chapter in the context of the stateless web API, is reused. 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 unary term `count` 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.

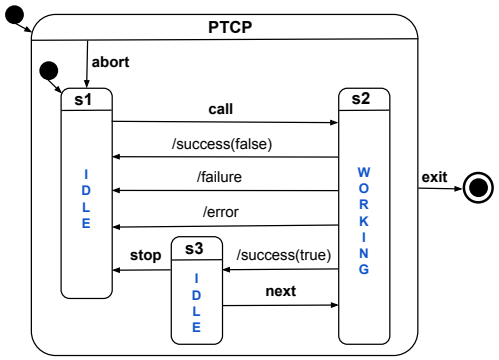

Fig. B.2 The complete PTCP protocol.

We need to enable a client to abort the execution of a goal:

```
ptcp(Pid, Target, Session) :-
    catch(state_1(Pid, Target, Session),
          '$abort_goal',
          ptcp(Pid, Target, Session)).
```

To make it work, the last line in the above implementation of `toplevel_spawn/2` must be changed into this:

```
    spawn(ptcp(Pid, Target, Session), Pid, Options).
```

Here is 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 using `toplevel_exit/2` (or just `exit/2` which as can be seen here means the same):

```
toplevel_exit(Pid, Reason) :-
    exit(Pid, Reason).
```

As suggested already in Chapter 1, programmers should not be burdened with having to remember the details of protocols and forms of built-in messages such as `'$call'`, `'$next'` and `'$stop'`. Instead, such details should be hidden behind interface predicates dealing with sending them, implementing `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'.
```

B.1.6 What is missing from the sketches?

The predicates implemented so far are sufficient for running many of the example programs given in Chapter 1 and Chapter 2 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 3, 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 3 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 handled 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.

Appendix C
A bigger example

Below, we have ported an Erlang program[1] into Web Prolog that uses seven concurrently running actor processes to solve the so called *Dining Philosophers problem*.[2] This program is also available and can be run in the tutorial accompanying the report.

```
sleep :-
    Time is random_float/10,
    sleep(Time).

doForks(ForkList) :-
    receive({
        {grabforks, {Left, Right}} ->
            subtract(ForkList, [Left,Right], ForkList1),
            doForks(ForkList1);
        {releaseforks, {Left, Right}} ->
            doForks([Left, Right| ForkList]);
        {available, {Left, Right}, Sender} ->
            (   member(Left, ForkList),
                member(Right, ForkList)
            ->  Bool = true
            ;   Bool = false
            ),
            Sender ! {areAvailable, Bool},
            doForks(ForkList);
        {die} ->
            format("Forks put away.~n")
    }).

areAvailable(Forks, Have) :-
    self(Self),
    forks ! {available, Forks, Self},
```

[1] https://github.com/acmeism/RosettaCodeData/blob/master/Task/Dining-philosophers/Erlang/dining-philosophers.erl

[2] https://en.wikipedia.org/wiki/Dining_philosophers_problem

```
    receive({
        {areAvailable, false} ->
            Have = false;
        {areAvailable, true} ->
            Have = true
    }).

processWaitList([], false).
processWaitList([H|T], Result) :-
    {Client, Forks} = H,
    areAvailable(Forks, Have),
    (   Have == true
    ->  Client ! {served},
        Result = true
    ;   Have == false
    ->  processWaitList(T, Result)
    ).

doWaiter([], 0, 0, false) :-
    forks ! {die},
    format("Waiter is leaving.~n"),
    diningRoom ! {allgone}.
doWaiter(WaitList, ClientCount, EatingCount, Busy) :-
    receive({
        {waiting, Client} ->
            WaitList1 = [Client|WaitList], % add to waiting list
            (   Busy == false,
                EatingCount < 2
            ->  processWaitList(WaitList1, Busy1)
            ;   Busy1 = Busy
            ),
            doWaiter(WaitList1, ClientCount, EatingCount, Busy1);
        {eating, Client} ->
            subtract(WaitList, [Client], WaitList1),
            EatingCount1 is EatingCount+1,
            doWaiter(WaitList1, ClientCount, EatingCount1, false);
        {finished} ->
            processWaitList(WaitList, R1),
            EatingCount1 is EatingCount-1,
            doWaiter(WaitList, ClientCount, EatingCount1, R1) ;
        {leaving} ->
            ClientCount1 is ClientCount - 1,
            flag(left_received, N, N+1),
            doWaiter(WaitList, ClientCount1, EatingCount, Busy)
    }).
```

```
philosopher(Name, _Forks, 0) :-
    format("~s is leaving.~n", [Name]),
    waiter ! {leaving},
    flag(left, N, N+1).
philosopher(Name, Forks, Cycle) :-
    self(Self),
    format("~s is thinking (cycle ~w).~n", [Name, Cycle]),
    sleep,
    format("~s is hungry (cycle ~w).~n", [Name, Cycle]),
    waiter ! {waiting, {Self, Forks}}, % sit at table
    receive({
        {served} ->
            forks ! {grabforks, Forks}, % grab forks
            waiter ! {eating, {Self, Forks}}, % start eating
            format("~s is eating (cycle ~w).~n", [Name, Cycle])
    }),
    sleep,
    forks ! {releaseforks, Forks}, % put forks down
    waiter ! {finished},
    Cycle1 is Cycle - 1,
    philosopher(Name, Forks, Cycle1).

dining :-
    AllForks = [1, 2, 3, 4, 5],
    Clients = 5,
    self(Self),
    register(diningRoom, Self),
    spawn(doForks(AllForks), ForksPid),
    register(forks, ForksPid),
    spawn(doWaiter([], Clients, 0, false), WaiterPid),
    register(waiter, WaiterPid),
    Life_span = 20,
    spawn(philosopher('Aristotle', {5, 1}, Life_span)),
    spawn(philosopher('Kant', {1, 2}, Life_span)),
    spawn(philosopher('Spinoza', {2, 3}, Life_span)),
    spawn(philosopher('Marx', {3, 4}, Life_span)),
    spawn(philosopher('Russel', {4, 5}, Life_span)),
    receive({
        {allgone} ->
            format("Dining room closed.~n")
    }),
    unregister(diningRoom),
    unregister(forks),
    unregister(waiter).
```