Invited Talk

Declarative Programming for Verification

Lessons and Outlook

Michael Leuschel

Lehrstuhl Softwaretechnik und Programmiersprachen
Institut für Informatik
University of Düsseldorf
leuschel@cs.uni-duesseldorf.de

Abstract

This paper summarises roughly ten years of experience using declarative programming for developing tools to validate formal specifications. More precisely, we present insights gained and lessons learned while implementing animators and model checkers in Prolog for various specification languages, ranging from process algebras such as CSP to model-based specifications such as Z and B.

Categories and Subject Descriptors I.2.3 [*Artificial Intelligence*]: Deduction and Theorem Proving—Logic Programming; D.2.4 [*Software Engineering*]: Software/Program Verification—Model Checking; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Mechancial verfication; Specification techniques

General Terms Languages; Verification

Keywords Logic Programming; formal methods; model checking; verification; animation.[1]

1. Introduction

A recurring theme within our research has been the use of declarative programming in general and Prolog in particular to both model and validate various high-level specification formalisms. This work started roughly ten years ago. At that time it was already well established that Prolog was a convenient language to model other languages, although some of the features such as co-routining where not yet fully appreciated (and probably still are not widely appreciated). Also, the idea of using (constraint) logic programming to analyse and validate programs and specifications only just started to become popular (leading, e.g., to workshops and special issues on verification and computational logic).[2]

[1] This research is partially supported by the EU funded FP7 project 214158: DEPLOY (Industrial deployment of advanced system engineering methods for high productivity and dependability).

[2] This situation has definitely changed now, and logic programming (in the form of Datalog) now has even appeared in the second version of the Dragon book [1] in the chapters on interprocedural dataflow analysis.

In this paper, I would like to convey the lessons that have been learnt during this undertaking. First, Section 2, discusses how various specification formalisms have been implemented (and parsed) in Prolog. We also briefly touch upon using Haskell for parsing. Section 3 then describes the implementation of various model checkers in Prolog. Finally, Section 4, summarises the lessons learned, and presents an outlook.

2. Modelling Specification Languages

2.1 CSP

CSP is a process algebra defined by Hoare [22]. The first semantics associated with CSP was a denotational semantics in terms of traces, failures and (failure and) divergences. Later an operational semantics was developed [41]. This semantics was the starting point of the implementation in [25]; one goal being to animate and validate (cf. Section 3.1) CSP specifications.

While Prolog is touted as a logic programming language, it is often impossible to translate logical inference rules or operational semantics rules directly in Prolog. Indeed, a literal translation often results in non-termination of the Prolog code, or, in case arithmetic built-ins are used, to calls which are not instantiated enough. That is why, e.g., [51] tries to statically infer a moding of the Z operators. This restricts the specifications which can be dealt with. Indeed, while for a program it may be reasonable to require well-moding, for specification this is often undesirable as it restricts the expressivity of the modeller. Further below we will show how we have overcome this limitation. But let us first turn to the problem of actually parsing in a CSP specification.

2.1.1 Experiences with DCGs

DCGs (definite clause grammars) are considered to be a convenient way of writing parsers within Prolog. Hence, in [25], we naturally implemented the parser using a DCG. However, our experience of using DCGs was mixed.

Indeed, to avoid constructing an inefficient, backtracking parser, one needs to carefully insert cuts. This issue is far from trivial and thus error-prone if done by hand. To keep matters under control we decided to adapt the CSP syntax, rather than supporting the existing (and sometimes very complex[3]) syntax [44] used by FDR [19]. In hindsight, this was a bad idea, as it discouraged CSP users and researchers from using our tool.

Also, while writing the context free grammar is simple in Prolog, writing a lexer (i.e., tokeniser) is not as straightforward. Hence, initially we encoded the whole parsing process within a single

[3] The reference implementation of the CSP-M parser [44] uses the bison parser generator, but the grammar, that actually serves as input for bison, is itself generated with a Perl script.

DCG. However, this obviously means that the grammar becomes less readable (mainly because whitespace and comments have to be dealt with) and it is also more difficult to get an efficient, predictive parser (as one token can consist of an arbitrarily long sequence of characters). We thus soon wrote a separate lexer in Prolog.

The bottom line is that DCGs are useful, but that they should be used in conjunction with a lexer (possibly generated using a tool, e.g., [39]). Also, it is difficult for a user to know where to put the cuts and to know if the parsing process will be deterministic. Hence, it does make sense to perform a separate analysis of the DCG (e.g., computing the nullable, first, follow sets from [1]), in order to know whether the grammar is suitable for predictive parsing and also to place the cuts according to the outcome of the analysis.

In [18] our interpreter was adapted for compensating CSP; the parser was now written using flex and bison. In a more recent version of our own tool, now supporting full machine readable CSP and being compliant with FDR, we have actually written the parser in Haskell and used the Parsec combinator parser library [23]. To allow interoperability, this new parser can deliver parse-results in different output formats, e.g., as a set Prolog facts or as a Java object representing the abstract syntax tree. Our parser was developed with the Glasgow Haskell Compiler (GHC) and is currently available as dynamic-link-library for four different architectures. No knowledge of Haskell is required to use the parser. Our practical experience, implementing a combinator parser for CSP-M in Haskell, was overall positive. The complete CSP-M syntax is specified in an single file `ParserCore.hs` with approximately 21KB of code (plus a separate lexer); which is still manageable. In total the parsing library contains about 4400 lines of Haskell. The specification of the CSP-M syntax is clearly separated from the generation of the results (for Prolog/Java/Haskell). The performance of our CSP-M parser is very satisfactory and it is also capable of parsing big auto generated files. For example, we used the parser to process a 3.5MB file which is the CSP encoding of a large labelled transitions system in about 22 seconds (on a 1.3GHz Pentium laptop).

2.1.2 Co-routining

The interpreter for CSP was obtained by translating the operational semantics rules from [41] into Prolog. The synchronisation between different CSP processes was achieved using unification. Another important aspect, which made a translation of the operational semantics possible, was the use of co-routining, as provided by several modern Prolog systems such SICStus Prolog [46], Ciao or now also SWI-Prolog.

Pure logic programs are independent of the selection rule [3, 33]. Co-routines (sometimes also called delay or wait declarations; first implemented in MU-Prolog by Naish [35]) build on this, by allowing the programmer to influence the selection rule via `when`, `block` or `freeze` declarations.[4] For a pure logic program, these declarations do not modify the logical meaning, but they can be used to improve the operational behaviour of the program. In particular, one can sometimes ensure that execution will terminate, or be much more efficient, in circumstances where Prolog's classical left-to-right selection rule would lead to non-termination or to unacceptable performance.

Take the following simple CSP specification, where `MAIN` is the process to be animated:

```
channel ch:Int
MAIN = ch?x:{0..99} -> ch!x -> STOP [| {|ch|} |]
       ch!55 -> ch?y:{1..100} -> STOP
```

[4] One lesson we learned very recently—thanks to Michael Hanus—is that the `block` declarations are much more efficient than `when` declarations. A rewriting of our interpreter resulted in a roughly 30 % performance increase.

Here `[|{|ch|}|]` is a synchronisation operator that forces all events on channel `ch` to synchronise. The process `ch?x:{0..99} -> ch!x -> STOP` first reads a value on the channel `ch`, provided the value is in the range of 0 to 99. It then outputs this value again on the same channel and then stops. Similarly, `ch!55 -> ch?y:{1..100} -> STOP` first outputs value 2 on the channel `ch` and then reads a value, provided the value is in the given range (and then stops).

Let us examine how to handle the synchronisation operator `[|{|ch|}|]` by looking at one of the inference rules from [41]:

$$\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{a} Q'}{P \parallel_X Q \xrightarrow{a} P' \parallel_X Q'} \ (a \in X)$$

Here we have P = `ch?x:{0..99} -> ch!x -> STOP`, Q = `ch!55 -> ch?y:{1..100} -> STOP`, a = `ch.55`, and X is the (event) closure of ch, i.e., $X = \{ch.0, ch.1, \ldots\}$. An interpreter now has a dilemma: either we first compute the outgoing transitions of P, but then we will compute 99 irrelevant transitions. Also, in general, an infinite number of solutions may exist. We could also compute first the outgoing transitions of Q. For this particular example, this would be the ideal order, as after that we only need to check whether P can perform the single event `ch.55`. However, at the next stage the roles of sender and receiver become reversed, and first computing the second process would be the bad choice. Also, in CSP more than two processes can synchronise, in which case possibly neither P nor Q would constrain the channel values.

We have overcome this problem by employing co-routining: basically, we first start to compute the outgoing transitions of P, but the value of x will be left uninstantiated (a free Prolog variable) and the membership test delays until the value is known. Afterwards we compute the outgoing transitions of Q, and via unification x will become instantiated to 55, immediately triggering the suspended membership test.

The following tiny interpreter, covering the three CSP constructs used above, captures the essence of our solution:

```
trans(out(X,T),X,T).
trans(in(X,L,T),X,T) :-
     when(ground(X),member(X,L)).
trans(sync(X,Y),A,sync(NX,NY)) :-
     trans(X,A,NX), trans(Y,A,NY).
```

With this interpreter we can compute the next two events of the above CSP process:

```
| ?- trans(sync(in(X,[0,...,99],out(X,stop)),
             out(55,in(Y,[1,...,100],stop))),E1,N1),
     trans(N1,E2,N2).
E1 = 55,
E2 = 55,
N1 = sync(out(55,stop),in(55,[1,55,100],stop)),
N2 = sync(stop,stop)  ? ;
no
```

Our full interpreter is obviously much more complex. Also, as it is possible that some channel fields are never given an explicit value, the full interpreter is wrapped into an outer layer which will enumerate any remaining uninstantiated channel fields at the very end.

In summary, the co-routines allowed a much more straightforward and still efficient translation of the operational semantics rules into Prolog.

The co-routines also allowed our tool to deal with certain specifications which cannot be dealt with by existing tools such as FDR. Indeed, it is a natural specification style is to have sub-processes which are infinite-state and only the global composition of the sub-processes makes the overall system finite state. For example, the following is a quite natural pattern, having an infinite state server process, but which is constrained by the environment so that the entire system (MAIN) is finite state. FDR cannot deal with this specification because the tool tries to expand and normalise the infinite state `Server` process *before* synchronising it with the `User` process. In our case, co-routining and unification, enable us to exhaustively animate the specification.

```
ID = {0..3}
channel new, ping, ack: ID
channel shutdown
Server =  new?id -> (Server ||| Serve(id)) []
          shutdown -> STOP
Serve(id) = ping?id -> ack!id -> Serve(id)
User = new?i -> UserActive(i)
UserActive(i) = ping!i -> ack!i -> UserActive(i)
MAIN = Server [| {| new,ping,ack,shutdown |} |] User
```

2.1.3 The Non-Ground Representation

While core CSP [22] is a relatively succinct language, full CSP-M as supported by FDR is a very extensive and expressive specification language. For example, CSP-M comprises a higher-order functional programming language and contains an extensive range of datatypes: booleans, integers, tuples, associative tuples, sequences, sets, and combinations thereof. Functions can be defined using complex pattern matching, even allowing combinations of patterns to be expressed using the `@@` operator. One unique aspect of CSP-M is the ability to use the concatenation operator ^ inside function patterns. E.g., one can define a function that computes the last element of a list by: `last(s^<x>) = x`.

To model the functional aspect of CSP-M, we have mainly used the non-ground representation [34, 21], i.e., representing a variable in the CSP-M specification by a Prolog variable. This enabled us to use Prolog's unification for pattern matching, as well as for synchronisations on channels already discussed above. The result is an efficient animator, which for some specifications is actually faster than FDR. The drawback is that in some circumstances non-declarative features had to be used (in particular to handle CSP-M's lambda expressions). Also, great care has to be taken not to accidentally instantiate variables of the object CSP-M specification. The use of the ground representation would have avoided these drawbacks, but at the cost of a much bigger and much slower interpreter.

2.2 B

About 10 years ago, the B-method was supported by industrial strength proving tools, such as Atelier-B [48] and the B-toolkit [5]. However, the only available animator (distributed with the B-toolkit) was not very user-friendly and required the user to guess the right values for operation arguments or non-deterministic choice variables.

One motivation behind the development of the PROB toolset [26, 27] was to overcome this restriction by implementing a fresh animator using logic programming, making animation more user-friendly but also enabling automated traversal of the state space of a specification.[5]

As was the case for CSP, co-routines turned out to be very useful.[6] Indeed, a core part of PROB is the kernel, which has been developed using co-routines. The kernel provides support for the B datastructures (sets, relations, functions, sequences, ...) and operations (relational composition, inverse, ...) on them. In fact, the kernel can be viewed as providing a constraint solver over B's datastructures, having some some similarities with the classical finite domain constraint solver CLP(FD) [11]. To make animation decidable, PROB requires finite base types in order to be able to enumerate, e.g., operation arguments and non-deterministic choice variables. The co-routines provide efficiency, by allowing one to replace a generate-and-test approach by test-and-generate and trying to fail as early as possible.

2.2.1 Coding Style

PROB is definitely the largest system we have developed. In order to be able to maintain it, it was essential to use an additional infrastructure to detect unexpected behaviours.

- we have developed a runtime checker, which can be turned off for performance, and which checks pre- and post-conditions of annotated predicates. It also checks that certain predicates do not fail or do not produce multiple solutions. The checker also contains a runtime type checker.

  This checker actually already started out in the code base for the partial evaluator ECCE [30], and has proven to be invaluable by catching and pinpointing unexpected behaviour.
- we use a large number of tests, ranging from unit tests for predicates to overall system regression tests.

  The unit tester also catches unexpected non-determinism and unexpected pending co-routines. The system regression tests also contain a large collection of mathematical theorems about set theory, relations, functions and sequences. They have actually uncovered a bug in SICStus Prolog (fixed in version 4.0.2), where our tool unexpectedly found a counter example for one such theorem.
- The above go hand-in-hand with an error manager module, which allows to easily report errors and unexpected behaviours. It ensures that the errors are brought to the attention of the user (depending on the mode in which the tool is run). This module was developed after several error messages had escaped attention (being simply printed on the terminal window).

The lesson is to spend some time on a proper checking and testing infrastructure; it will be worth in the long run. Some features, that would have helped us even further, is a proper integrated development environment for Prolog, with support for navigation, source code highlighting and refactoring (e.g., adding arguments to predicates, moving predicates into other modules, detecting unreachable code,...). Some progress is made into that direction [45]. We are also trying to build ourselves an Eclipse environment for Prolog development, based upon our editor BE4 [6].

2.3 More Related Work

Prolog has been a popular programming language to implement other programming or specification languages. A well-known example is Erlang, which was initially implemented in 1986 using a Prolog interpreter [4]. A logic programming approach to encode denotational semantics specifications was applied in [24] to verify an Ada implementation of the "Bay Area Rapid Transit" controller. On the side of specification languages, many formalisms have been implemented using Prolog. To mention but a few, there is the Verilog animator [9] by Bowen, as well as animators for subsets of Z such as [20], or [51]. However, none of these used any of the advanced features such as tabling, constraints, co-routining. Indeed, these new and exciting features are not well known outside of the core logic programming community. Even many within the community are not aware of all of the benefits that these improvements provide.

[5] Unknowingly to us, researchers from Besançon were pursuing a similar avenue, which lead to the BZTT [8, 2] tool.

[6] Unification also turned out to be useful, as it later enabled us to extend our tool to support combined CSP and B specifications [10].

3. Model Checkers

Model checking [13] is a well known technique for the validation and verification of correctness properties of hardware, and more recently software systems.

3.1 CTL

In [31] we did implement a CTL model checker using the XSB-Prolog system [43, 42] with its support for tabling [12]. Tabling allowed us to write a complete interpreter for CTL formulas, implemented as a pure logic program. This enabled certain analysis and transformation tools to be applied (which required pure logic programs), even achieving some infinite state model checking [29, 17].

The model checker was also very flexible, in that it could be applied to other formalisms simply by writing Prolog predicates for `trans/3`, `prop/2`, and `start/1`.

3.1.1 Speed of tabling

XSB is very convenient for model checking. Other work has already realised the usefulness of XSB for program analysis [14]. It often enables the programmer to write an efficient, declarative version of a program, which in a classical Prolog system would have required a more imperative solution with loop checking and fixpoint detection.

On top of that, tabling is often surprisingly fast. On some examples, the XSB model checker was outperforming a similar model checker encoded in SICStus Prolog (maintaining the table as an asserted dynamic fact database) by a factor of 400. As it turned out, the big difference was not so much to the tabling but to the improved (multi-argument) indexing that XSB uses for tables.

Let us first perform a simple experiment, comparing XSB tables against SICStus Prolog dynamic facts. First, let us examine the XSB-Prolog program:

```
b :- time(bench(1000000)), time(bench(1000000)).
bench(0).
bench(X) :- X>0, p(X), X1 is X-1, bench(X1).

:- table p/1.
p(X).
```

The first call to `bench` adds a million new entries to the table (after checking whether they are already in the table). This took 0.861 seconds CPU time with XSB 3.1 on a MacBook Pro with a 2.33 GHz Core2 Duo processor. The second call to `bench` no longer adds to the table, there are “only” a million lookups in the table. This call was marginally faster at 0.837 seconds.

The above code can be translated into classical Prolog as follows (this only translates the aspect of memorising which calls to `p/1` were made; it does not address the more difficult issue of propagating computed answers, which we ignore here):

```
b :- time(bench(1000000)), time(bench(1000000)).
bench(0).
bench(X) :- X>0, p(X), X1 is X-1, bench(X1).

p(X) :- check_table(X).

:- dynamic table/1.
check_table(X) :-
          table(X) -> true ; assert(table(X)).
```

The first call to `bench` took 2.43 seconds with SICStus 4.0.2, the second call took only 0.21 seconds. Hence, XSB is faster when adding to a table and SICStus is faster when checking for existing entries. The differences are marked, but not as dramatic so as to explain a factor 400 speed difference for model checking applications.

If we replace `p(X)` by `p(1,X)` in the above examples, XSB still takes almost the same time. SICStus Prolog now, however, takes considerably longer: already checking and asserting 100,000 new facts takes 403.65 seconds and then re-checking them takes 404.78 seconds.

In conclusion, looking up in a table in XSB is actually slower than looking up in a dynamically asserted predicate in SICStus; adding to a table in XSB is faster than asserting facts in SICStus; a big speed difference appears when XSB can make use of its multi-argument indexing for tables (implemented via tries)!

One lesson is that XSB is very fast and flexible. Another lesson is that it can be possible to emulate tabling using asserted facts with reasonable speed (at least without computed answer propagation), but one has to be very careful to ensure that first-argument indexing can be used to locate the table entries.

3.1.2 Infinite Answers and Negation

In general tabling improves termination. It is maybe surprising that this is not always the case. Let us examine the following example, which captures the essence of what happens when there are infinitely many counter example traces in our model checker (which is very common):

```
p(X) :- p2(X,_L).

:- table p2/2.
p2(a,[]).
p2(X,[X|T]) :- p2(X,T).
```

Calling `p(X)` in a classical Prolog will give an answer ($X = a$), but the same call will loop infinitely in XSB in the default local scheduling mode. Fortunately, one can compile XSB to use so-called batched mode rather than local scheduling. With this version of XSB one does get the first answer $X = a$ for `p(X)`.[7]

Still, even in batched mode, there is a problem if one uses negation on the above query: here classical Prolog would terminate, but XSB does not:

```
:- table q/0.

:- table p/1.

q :- tnot(p(a)).
```

One solution is to compile XSB with certain flags set (demand evaluation). However, this mode of XSB did not always work in our experiments, and in the latest release (3.1) we were not able to compile XSB with the required flags. The bottom-line was that our CTL model checker was thus practically useless for more complicated formulas involving negation (and many temporal operators do require negation at some point).

We would like to see a version of XSB that can deal with negated calls having an infinite number of answers. In addition, it would be useful to have support for co-routining in XSB (even if the system required that there should be no pending goals when a tabled call is reached), so that one could apply the model checker to our interpreters from Section 2.

3.1.3 Partial Evaluation

Due to the declarative nature of our CTL model checker, it was very straightforward to apply partial evaluation tools to specialise

[7] However, trying to find another answer will result in an infinite loop. With classical Prolog one gets the answer $X = a$ infinitely often.

the model checker for a particular formula and/or a particular interpreter and system. This worked extremely well for Petri nets [29] and Object Petri nets [17]. We were even able to use the online partial evaluator ECCE [30] to perform certain infinite model checking tasks, and a similarity between online partial evaluation and algorithms from Petri net theory were uncovered [29, 28].

Unfortunately, these results do not yet apply when using the much more involved interpreters for CSP-M or B described earlier in Section 2. One hurdle was the integration of the specialised model checker into a running system. Another, was scalability of the partial evaluator to larger Prolog programs, using, e.g., modules and co-routines. The lesson is that existing partial evaluation tools are not yet ready to be applied to real-life applications. One solution to our problems would be the development of a just-in-time specialiser in the style of [40] for Python, which transparently specialises the program during execution. We are also working towards ensuring that existing classical specialisers scale to larger programs and can be integrated with, e.g., PROB.

3.2 LTL and Symmetry Reduction

The lesson we learned from our experiences with the CTL model checker was to manage our own tables, using asserted facts, taking special care that we used good hashing functions to obtain good first argument indexing. As far as hashing is concerned, we also learned that the SICStus Prolog `term_hash/2` predicate generates more collisions than one may expect. Also, unfortunately, it only works on ground terms. We thus developed our own hashing predicate.

We also realised that CTL was a less useful temporal logic than LTL (see [50]). Hence, we started coding an LTL model checker in Prolog, using the tableaux algorithm from [13].

Unfortunately, the speed was somewhat disappointing and we actually moved to coding the model checker in C, interfacing with our various interpreters using the foreign language interface of SICStus Prolog. This resulted in substantial speed improvements (1-2 orders of magnitude depending on formula) and a practically useful validation tool [32]

We have also successfully used the foreign language interface to obtain efficient symmetry reduction via graph canonicalisation using the tool NAUTY written in C [47]. This was again after a first version of the same approach in Prolog, whose performance turned out to be rather disappointing [49].

The lesson is that some algorithms, especially on graphs, can be encoded much more efficiently in C, but that the foreign language interfaces are stable and reliable enough to enable a good integration of C and Prolog code.

We have also used various other foreign language interfaces (to Tcl/Tk and to Java) to good effect, for example to provide a graphical user interface to our tools. This approach very naturally imposes a clean separation between graphical user interface and the application logic.

3.3 More Related Work

There are variety of relatively generic CTL model checkers, such as [31, 36, 16]. Both [36] and [16] are based on constraint logic programming. [36] requires constructive negation, and as such only a prototype implementation seems to exist. [16] is tailored towards verification of infinite state systems.[8] We also would like to mention the work in [7], which mapped Petri nets to CLP programs for verification. The CTL model checker in [31] is generic, and can be applied to any specification language that can be encoded in Prolog. Unfortunately, the model checker relies on tabling, and as such it can only run on XSB Prolog [42], which does not support co-routines and hence the system can *not* be applied to our interpreters for CSP and B (and hence also Z). The same can be said for the pure LTL model checker from [37] or the XMC system [15, 38] for the modal mu-calculus and value-passing CCS.

[8] Unfortunately, the corresponding DMC prototype available at `http://www.disi.unige.it/person/DelzannoG/DMC/dmc.html` no longer runs on current SICStus Prolog versions and the code is no longer maintained.

4. Discussion and Conclusion

In summary, some of the lessons we learned were the following:

- Prolog's co-routines can be very useful and are probably still underrated. They can be very useful to translate operational semantics rules in a natural way, and they can be used to write one's own constraint solver.
- DCGs can be tricky to use. If you use them, be sure to use a lexer and try to analyse the grammar to determine whether it is suitable for predictive parsing.
- Tabling can be very efficient and convenient, but is only available on a single Prolog system (XSB) which has limitations. Especially in the presence of negation can be problematic, and the absence of convenient co-routining is a major drawback.
- Foreign language interfaces are very reliable and can be immensely useful to encode graphical user interfaces (e.g., in Tcl/Tk or Java) or certain critical algorithms which can be implemented more efficiently in an imperative language (e.g., in C).
- Systematic testing and runtime checking can help build a large but still maintainable system in Prolog.
- Partial evaluation is not yet ready to be applied to larger applications, but progress is being made.
- Sometimes it is good to view Prolog as a dynamic language, and not feel guilty about using the non-ground representation or dynamically asserting or retracting facts. In many circumstances taking these shortcuts will lead in much shorter and faster code, and it is not clear whether the effort in attempting to write a declarative version would be worthwhile.

In summary, we believe that for certain tasks declarative programming is very well suited, and can produce more efficient implementations more quickly than other formalisms. One such task is validation of high-level specification languages. We believe that developing an animator or model checker for full CSP-M or full B in an imperative programming language would have required considerably more resources. Also, the end result would probably have been less efficient than our existing Prolog implementation.

Acknowledgments

I would like to thank the co-authors of the papers that are behind this work: Jens Bendisposto, Michael Butler, Marc Fontaine, Helko Lehmann, Thieryy Massart, Daniel Plagge, Corinna Spermann, Edd Turner.

References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers. Principles, Techniques, and Tools (Second Edition)*. Addison Wesley, 2007.

[2] F. Ambert, F. Bouquet, S. Chemin, S. Guenaud, B. Legeard, F. Peureux, M. Utting, and N. Vacelet. BZ-testing-tools: A tool-set for test generation from Z and B using constraint logic programming. In *Proceedings of FATES'02, Formal Approaches to Testing of Software*, pages 105–120, August 2002.

[3] K. R. Apt. Introduction to logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 10, pages 495–574. North-Holland Amsterdam, 1990.

[4] J. Armstrong. A history of Erlang. In B. G. Ryder and B. Hailpern, editors, *HOPL*, pages 1–26. ACM, 2007.

[5] B-Core (UK) Ltd, Oxon, UK. *B-Toolkit, On-line manual*, 1999. Available at `http://www.b-core.com/ONLINEDOC/Contents.html`.

[6] J. Bendisposto and M. Leuschel. BE4: The B extensible eclipse editing environment. In *Proceedings of the 7th International B Conference (B2007)*, LNCS 4355, pages 270–273, Besancon, France, 2007. Springer-Verlag.

[7] B. Bérard and L. Fribourg. Reachability analysis of (timed) Petri nets using real arithmetic. In *Proceedings of Concur'99*, LNCS 1664, pages 178–193. Springer-Verlag, 1999.

[8] F. Bouquet, B. Legeard, and F. Peureux. CLPS-B - a constraint solver for B. In J.-P. Katoen and P. Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 2280, pages 188–204. Springer-Verlag, 2002.

[9] J. Bowen. Animating the semantics of VERILOG using Prolog. Technical Report UNU/IIST Technical Report no. 176, United Nations University, Macau, 1999.

[10] M. Butler and M. Leuschel. Combining CSP and B for specification and property verification. In *Proceedings of Formal Methods 2005*, LNCS 3582, pages 221–236, Newcastle upon Tyne, 2005. Springer-Verlag.

[11] M. Carlsson and G. Ottosson. An open-ended finite domain constraint solver. In H. G. Glaser, P. H. Hartel, and H. Kuchen, editors, *Proc. Programming Languages: Implementations, Logics, and Programs*, LNCS 1292, pages 191–206. Springer-Verlag, 1997.

[12] W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, January 1996.

[13] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[14] M. Codish, B. Demoen, and K. F. Sagonas. Xsb as the natural habitat for general purpose program analysis. In *ICLP*, page 416, 1997.

[15] B. Cui, Y. Dong, X. Du, N. Kumar, C. R. Ramakrishnan, I. V. Ramakrishnan, A. Roychoudhury, S. A. Smolka, and D. S. Warren. Logic programming and model checking. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Proceedings of ALP/PLILP'98*, LNCS 1490, pages 1–20. Springer-Verlag, 1998.

[16] G. Delzanno and A. Podelski. Constraint-based deductive model checking. *STTT*, 3(3):250–270, 2001.

[17] B. Farwer and M. Leuschel. Model checking object Petri nets in Prolog. In *Proceedings PPDP '04*, pages 20–31, New York, NY, USA, 2004. ACM Press.

[18] C. Ferreira and M. Butler. A process compensation language. In T. Santen and B. Stoddart, editors, *Proceedings Integrated Formal Methods (IFM 2000)*, LNCS 1945, pages 424–435. Springer-Verlag, November 2000.

[19] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement — FDR2 User Manual (version 2.8.2)*.

[20] M. A. Hewitt, C. O'Halloran, and C. T. Sennett. Experiences with piza, an animator for z. In *ZUM '97: Proceedings of the 10th International Conference of Z Users on The Z Formal Specification Notation*, pages 37–51, London, UK, 1997. Springer-Verlag.

[21] P. Hill and J. Gallagher. Meta-programming in logic programming. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, pages 421–497. Oxford Science Publications, Oxford University Press, 1998.

[22] C. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[23] G. Hutton and E. Meijer. Monadic Parser Combinators. Technical Report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham, 1996.

[24] L. King, G. Gupta, and E. Pontelli. Verification of a controller for BART. In V. L. Winter and S. Bhattacharya, editors, *High Integrity Software*, pages 265–299. Kluwer Academic Publishers, 2001.

[25] M. Leuschel. Design and implementation of the high-level specification language CSP(LP) in Prolog. In I. V. Ramakrishnan, editor, *Proceedings of PADL'01*, LNCS 1990, pages 14–28. Springer-Verlag, March 2001.

[26] M. Leuschel and M. Butler. ProB: A model checker for B. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.

[27] M. Leuschel and M. Butler. ProB: an automated analysis toolset for the B method. *STTT*, 10(2):185–203, 2008.

[28] M. Leuschel and H. Lehmann. Coverability of reset Petri nets and other well-structured transition systems by partial deduction. In J. Lloyd, editor, *Proceedings of the International Conference on Computational Logic (CL'2000)*, LNAI 1861, pages 101–115, London, UK, 2000. Springer-Verlag.

[29] M. Leuschel and H. Lehmann. Solving coverability problems of Petri nets by partial deduction. In M. Gabbrielli and F. Pfenning, editors, *Proceedings of PPDP'2000*, pages 268–279, Montreal, Canada, 2000. ACM Press.

[30] M. Leuschel, B. Martens, and D. De Schreye. Controlling generalisation and polyvariance in partial deduction of normal logic programs. *ACM Transactions on Programming Languages and Systems*, 20(1):208–258, January 1998.

[31] M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialisation. In A. Bossi, editor, *Proceedings LOPSTR'99*, LNCS 1817, pages 63–82, Venice, Italy, 2000.

[32] M. Leuschel and D. Plagge. Seven at a stroke: LTL model checking for high-level specifications in B, Z, CSP, and more. In Y. Aït-Ameur, F. Boniol, and V. Wiels, editors, *Proceedings Isola 2007*, Revue des Nouvelles Technologies de l'Information RNTI-SM-1, pages 73–84, 2007.

[33] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.

[34] B. Martens and D. De Schreye. Why untyped non-ground meta-programming is not (much of) a problem. *The Journal of Logic Programming*, 22(1):47–99, 1995.

[35] L. Naish. An introduction to MU-Prolog. Technical Report 82/2, Department of Computer Science, University of Melbourne, Melbourne, Australia, March 1982 (Revised July 1983).

[36] U. Nilsson and J. Lübcke. Constraint logic programming for local and symbolic model checking. In J. Lloyd, editor, *Proceedings of the International Conference on Computational Logic (CL'2000)*, LNAI 1861, pages 384–398, London, UK, 2000. Springer-Verlag.

[37] R. L. Pokorny and C. R. Ramakrishnan. Model checking linear temporal logic using tabled logic programming. In *Proceedings Tabling in Parsing and Deduction TAPD 2000*, Vigo, Spain, September 2000.

[38] Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In O. Grumberg, editor, *Proceedings CAV'97*, LNCS 1254, pages 143–154. Springer-Verlag, 1997.

[39] P. B. Reintjes and S. Rajgopal. Multi/plex: Prolog tools for formal languages. In *LPE*, pages 81–87, 1993.

[40] A. Rigo. Representation-based just-in-time specialization and the Psyco prototype for Python. In N. Heintze and P. Sestoft, editors, *PEPM*, pages 15–26. ACM, 2004.

[41] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1999.

[42] K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 442–453, Minneapolis, Minnesota, May 1994. ACM.

[43] K. Sagonas, T. Swift, D. S. Warren, J. Freire, P. Rao, B. Cui, E. Johnson, L. de Castro, R. F. Marques, S. Dawson, and M. Kifer. *XSB Programmer's Manual*. Available at `http://xsb.sourceforge.net/`.

[44] J. B. Scattergood. *Tools for CSP and Timed-CSP*. PhD thesis, Oxford University, 1997.

[45] A. Serebrenik, T. Schrijvers, and B. Demoen. Improving prolog programs: Refactoring for prolog. *Theory and Practice of Logic Programming*, 8:201–215, 2008.

[46] S. SICS, Kista. *SICStus Prolog User's Manual*. Available at `http://www.sics.se/sicstus`.

[47] C. Spermann and M. Leuschel. ProB gets nauty: Effective symmetry reduction for B and Z models. In *Proceedings Symposium TASE 2008*. IEEE, June 2008. to appear.

[48] F. Steria, Aix-en-Provence. *Atelier B, User and Reference Manuals*, 1996. Available at `http://www.atelierb.societe.com`.

[49] E. Turner, M. Leuschel, C. Spermann, and M. Butler. Symmetry reduced model checking for B. In *Proceedings Symposium TASE 2007*, pages 25–34, Shanghai, China, June 2007. IEEE.

[50] M. Y. Vardi. Branching vs. linear time: Final showdown. In T. Margaria and W. Yi, editors, *TACAS'01*, LNCS 2031, pages 1–22. Springer, 2001.

[51] M. Winikoff, P. Dart, and E. Kazmierczak. Rapid prototyping using formal specifications. In *Proceedings of the 21st Australasian Computer Science Conference*, pages 279–294, Perth, Australia, February 1998.