Animation and Model Checking of CSP and B using Prolog Technology

Michael Leuschel, Laksono Adhianto, Michael Butler, Carla Ferreira, and Leonid Mikhailov

Department of Electronics and Computer Science
University of Southampton
Highfield, Southampton, SO17 1BJ, UK
mal@ecs.soton.ac.uk
Fax: +44 2380 59 3045

Abstract. We describe practical experiences of using a logic programming based approach to model and reason about critical systems. We argue that logic programming with co-routining, constraints, and tabling is a good foundation for developing, animating, and model checking new specification languages. We present animators and model checkers currently being developed for two different extensions of CSP and for the B method.
Keywords: Specification, Verification, Model Checking, Concurrency, B-method, Implementation, Compilation, Partial Evaluation.

1 Introduction

This paper describes work carried out within the projects ABCD[1], iMoc[2], and Matisse[3]. Our objective is to increase the uptake of formal methods in the business critical systems industry by lowering the cost of entry and increasing the benefits of using formal modelling. The focus is on support of system definition and architectural design so that the systems integrator can more easily model systems and validate proposed system architectures. A main point is to apply formal methods—such as model checking—early on in the software development cycle, i.e., on high-level specifications or high-level prototypes of the particular business critical system to be developed.

Currently we are focussing our efforts on providing support for the CSP-formalism [13] to express the concurrent aspects of a critical system and the B-method [1] to represent the state-based aspects. We are planning to combine these two methodologies into one [3], as well as provide integration with UML.

[1] "Automated validation of Business Critical systems using Component-based Design," EPSRC grant GR/M91013, with industrial collaborators IBM, ICL, Praxis Critical Systems, and Roke Manor Research.

[2] "Infinite state MOdel Checking using partial evaluation and abstract interpretation," EPSRC grant GR/N11667.

[3] European Commission IST Programme RTD Research Project IST-1999-11435.

In the remainder of this paper we show how the latest generation Prolog technology (co-routining, constraints, tabling, partial evaluation) is helping us to develop animation and verification tools for these methodologies in a flexible but also efficient manner.

2 Interpretation and Animation of CSP

CSP is a process algebra defined by Hoare [13]. Elementary CSP is not very useful in practice, either as a programming or a specification language, because of its lack of value passing and the absence of more refined operators. Hence [23, 11, 24] present an extension of CSP (which we henceforth call CSP_M), along with a machine-readable ASCII syntax. CSP_M and the tools FDR and PROBE are used by industry, especially in the security area [22]. In this extension of CSP one can:

- pass tuples of datavalues on channels. For example, $a!1 \rightarrow STOP$ will output the datavalue 1 on the channel a, while $a?x \rightarrow P(x)$ will read a datavalue on channel a and bind x to the value read. One can also put additional constraints on values that are received on a channel, as in $a?x : x > 2 \rightarrow P(x)$.
- use constructs such as the if-then-else, let-constructs, and operators such as interrupt and timeout.
- use sets and set operations, sequences and sequence operations, integers and arithmetic operators, tuples, enumerated types, ranges, etc.

This extension of CSP was heavily influenced by functional programming languages, and hence relies on pattern matching as a means to synchronise on channels. [15] showed how, by using logic rather than functional programming as a foundation, one can use unification to synchronise on channels," thus leading to a more powerful[4] language CSP(LP) which to some extent reconciles CSP_M with (concurrent) logic programming languages. The basic syntax of CSP(LP) is summarised in Figure 1.

Implementation as a logic program with co-routining The implementation of CSP(LP) described in [15] basically encodes the operational semantics of [23]; the main difference lies in the generalisation of the synchronisation mechanism.

The co-routining capabilities of SICStus Prolog turned out to be essential to easily implement CSP(LP). (This was not really explained in [15].) Take for example the CSP(LP) expression `a?_x:_x>1 -> STOP  [| {a} |] a!3 -> STOP`. When the interpreter reaches the expression `a?_x:_x>1 -> STOP` it does not yet know the value of `_x` and hence cannot yet evaluate the test `_x>1`. Co-routining allows us to suspend this test (e.g., using the call `when(ground(_x),_x>1))` until the value for `_x` is provided by `a!3 -> STOP`. Without co-routining the

[4] Whether the extensions provided by CSP(LP) is really useful in practice is still under consideration. But one can always use CSP(LP) just like CSP_M.

<table>
<tr><th>Operator</th><th>Syntax</th><th>Ascii Syntax</th></tr>
<tr><td>stop</td><td>$STOP$</td><td><code>STOP</code></td></tr>
<tr><td>skip</td><td>$SKIP$</td><td><code>SKIP</code></td></tr>
<tr><td>prefix</td><td>$a \rightarrow Q$</td><td><code>a->P</code></td></tr>
<tr><td>conditional prefix</td><td>$a?x : x > 1 \rightarrow P$</td><td><code>a?_x:_x>1->P</code></td></tr>
<tr><td>external choice</td><td>$P \Box Q$</td><td><code>P [] Q</code></td></tr>
<tr><td>internal choice</td><td>$P \sqcap Q$</td><td><code>P |~| Q</code></td></tr>
<tr><td>interleaving</td><td>$P|||Q$</td><td><code>P ||| Q</code></td></tr>
<tr><td>parallel composition</td><td>$P \llbracket A \rrbracket Q$</td><td><code>P [| A |] Q</code></td></tr>
<tr><td>sequential composition</td><td>$P; Q$</td><td><code>P ->> Q</code></td></tr>
<tr><td>hiding</td><td>$P \backslash A$</td><td><code>P \\ A</code></td></tr>
<tr><td>renaming</td><td>$P[R]$</td><td><code>P [[R]]</code></td></tr>
<tr><td>timeout</td><td>$P \triangleright Q$</td><td><code>P [> Q</code></td></tr>
<tr><td>interrupt</td><td>$P \triangle_i Q$</td><td><code>P /\ Q</code></td></tr>
<tr><td>if then else</td><td>$if\ t\ then\ P\ else\ Q$</td><td><code>if T then P else Q</code></td></tr>
<tr><td>let expressions</td><td>$let\ v = e\ in\ P$</td><td><code>let V=E in P</code></td></tr>
<tr><td>agent definition</td><td>$A = P$</td><td><code>A = P;</code></td></tr>
</table>

Fig. 1. Summary of syntax of CSP(LP)

interpreter would have become much more intricate, probably considerably extending our development time and also making the interpreter less amenable to techniques such as partial evaluation (see Section 3).

Animation An animator was developed in [15] using the Tcl/Tk library of SICStus Prolog 3.8. The tool is being refined and was inspired by the ARC tool [12] for system level architecture modelling and supports (backtrackable) step-by-step animation of the CSP specification as well as random animation, and an iterative deepening search for invalid states.

All of this provides verification assistance, which is also applicable if the CSP(LP) description is infinite state. Unfortunately the verification is always going to be *non-exhaustive*. In the next section we show how we have implemented *exhaustive* model checking for finite state CSP(LP) specifications.

3 Model Checking using Tabling and Partial Evaluation

Given a logic programming representation for a system, one can try to apply various logic programming based model checking approaches such as [21, 7], [9], [18, 17] [20]. Our first attempts have focused on using the CTL intepreter described in [18]. This interpreter has to be executed by a table-based Prolog system (such as XSB-Prolog) to achieve exhaustive CTL model checking for all finite state systems. A drawback of the interpreter in [18] is that it does not provide counter examples if the formula to be checked is false or witnesses if the formula to be checked is true. We have therefore extended the CTL interpreter to provide counter examples and witnesses (for some cases at least). An advantage of [18] is that the interpreter is fully declarative and also amenable to optimisation techniques such as partial evaluation.

Now, when trying to combine the CTL interpreter with the CSP(LP) interpreter in [15] we are faced with a technical problem. The CTL interpreter requires tabling and the CSP(LP) interpreter requires co-routining (future versions might also require constraints). No Prolog system, as of today, provides a full integration of both (e.g., XSB Prolog does not support co-routining using `when`; it supports attributed variables but the integration with the tabling mechanisms is still problematic).

However, both the CSP(LP) and the CTL interpreter are fully declarative and amenable to optimisation techniques such as partial evaluation. One solution then is to partially evaluate the interpreters put together, specialising them for the particular CSP(LP) program and the particular CTL property, thereby trying to specialise the co-routining primitives away. This actually works for a large class of systems, and our hope is that it will lead to efficient model checkers for a certain class of systems.

To get this approach to work we had to extend the LOGEN system [16] to:

- handle tabling directives, i.e., generate proper tabling directives for those predicates in the residual program which are derived from a tabled predicate in the original program
- properly handle tabled negation(s) `tnot` as well as `sk_not`.

Using this improved LOGEN system, we were able to model check a certain number of CSP(LP) specifications. We have integrated all of this into the animator (see Figure 2), so that both LOGEN and XSB-Prolog are called transparently for the user.

We are still in the process of evaluating the practicality of the approach. For some CSP(LP) systems the approach seems to fare very well. It is, however, already clear that this approach is not a solution in general: sometimes the only way to get rid of the co-routining is to provide the full description of the initial state to the partial evaluator and to ensure that the partial evaluator essentially generates the whole state space. Even using a fast offline specialisation system such as LOGEN [16] this is going to lead to unsatisfactory performance: 1) memoisation (detecting configurations that have already been encountered) in LOGEN (or other partial evaluators) does not use the sophisticated data structures and algorithms which contribute to the efficiency of tabling in XSB Prolog. 2) The generated residual program is very large, and just consulting this residual program (to do the final model checking) can take a considerable amount of time. A possible way to lift this restriction is to re-code LOGEN in XSB Prolog so that it makes use of the efficient XSB tabling mechanism for memoisation.

4 Animation of the StAC Language

In collaboration with IBM, we have developed an extension of CSP which provides support for compensation operations as required, e.g., in transaction processing applications. This led to the StAC language [10]. It turned out that we could port the technology developed for CSP(LP) to StAC, providing us with verification tools with relatively little effort (see Figure 4 below).

The distinctive characteristic about StAC is the concept of compensation. This concept can have many interpretations, the most common being recovering

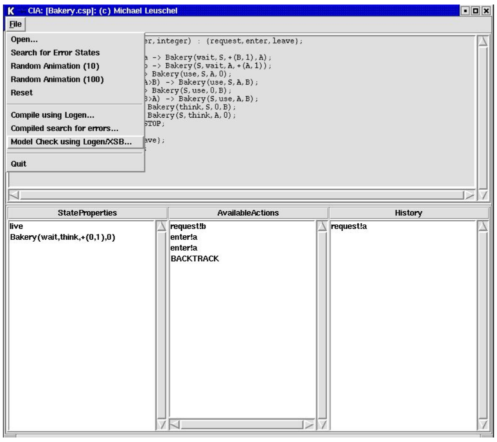

Fig. 2. The CSP(LP) Animator

from an error. An example of the use of the compensation operators is presented in [10].

A process in StAC will be decomposed into several sub-processes in a top-down approach. At the bottom level there will only be activities (each activity is an atomic computation), so they can not be further decomposed and we used the B notation [1] to specify these activities. The syntax of StAC is presented in Figure 3.

Operator	Syntax	Ascii Syntax
activity	A	`A`
skip	0	`0`
condition	$b \rightarrow P$	`b -> P`
recursion	$rec(N)$	`rec(N)`
sequence	$P; Q$	`P ; Q`
parallel	$P \parallel Q$	`P || Q`
choice	$P[]Q$	`P [] Q`
compensation pair	$P \div Q$	`P % Q`
compensate	$\boxtimes$	`<*>`
commit	$\boxdot$	`<>`
compensation scoping	$[P]$	`[P]`

Fig. 3. Syntax of StAC

In the conditional operator, process P is guarded by a boolean function b. The sequence $P; Q$ determines that P must be executed before Q. In parallel process $P \parallel Q$, the execution of the activities of P and Q is interleaved. The choice $P[]Q$ selects whichever of P or Q is enabled. If both P and Q are enabled, the choice is made by the environment.

The next operators are related to the compensation concept. The compensation pair $P \div Q$ expresses that P has Q as its compensation process. The compensation process is constructed in the reverse order of process execution, for example, $(A \div A'); (B \div B')$ behaves as $A; B$ and has the compensation process $B'; A'$. A compensation process can be viewed as a stack where processes are pushed into the top of the stack.

The compensate operator $\boxtimes$ causes the compensation process to be executed. The process $(A \div A'); \boxtimes$ behaves as $A; A'$, first A is executed and A' is added to the compensation process, next $\boxtimes$ causes the compensation process A' to be executed. The commit operator $\boxdot$ clears the compensation process, after a commit the compensation process is set to 0.

The compensation scoping operator $[P]$ creates a new compensation process within the square brackets. In the process $(A \div A'); [\,(B \div B')\,]$ the compensation process within the brackets is just B', and A' is excluded because its outside the brackets. If we added the compensate operator inside the brackets, $(A \div A'); [\,(B \div B'); \boxtimes\,]$, the overall process would behave as $A; B; B'$, since the $\boxtimes$ just executes the compensation process within the brackets. Another feature of the compensation scoping operator is that compensation is remembered if compensate is not performed. The process $(A \div A'); [\,(B \div B')\,]; (C \div C')$ has process $C'; B'; A'$ as its compensation process, which includes the compensation process B' defined inside the brackets. B' is retained because there is no commit within the brackets.

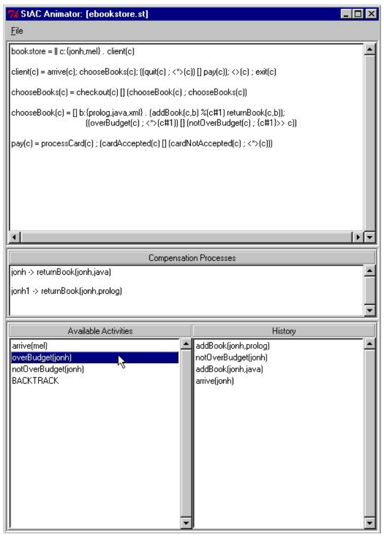

Fig. 4. The StAC Animator running on an electronic bookstore

StAC is a typical domain specific specification language, which evolved during the development process. Our experience has shown that the latest generation Prolog systems are very well suited to prototype and develop such languages.

5 Animation and Model Checking of B

The B-method, originally devised by J.-R. Abrial [1], is a powerful methodology for developing safety critical systems. It is based on the notions of an abstract machine and on refinement. It is supported by industrial strength tools, such as B-toolkit [2] and Atelier B [25]. However, the animation facilities of these tools leave something to be desired, and the tools also lack certain verification capabilities such as model checking (neither temporal logic based [6] nor model based [14] are supported). We are therefore developing our own tool which provides these capabilities.

The first problem to be overcome is the translation of specifications written in abstract machine notation (AMN) [1] into a notation convenient for interpretation within Prolog. AMN notation is a formal specification notation used in the B-method. Due to the complexity of the AMN specification language and the fact that it was not originally developed with the computer automation in mind, developing an interpreter for B specifications represents a considerable task in itself.

Two initial attempts at the development of the parser/compiler were undertaken using JavaCC (a general purpose compiler constructor - [27]) and JavaLIL (a tool, developed internally in the department, for converting specifications written in Z or AMN notations into HOL or PVS representations - [8]) were not successful. The problems can be attributed in the case of JavaCC to the complexity of AMN grammar and in the second case to the lack of proper documentation of the internals of JavaLIL.

The current development effort is facilitated by the JBTools package develped by Tatibouet [26] and the Pillow package [4]. The JBTools package permits translation of AMN specifications into the XML language, while the Pillow package allows for translation of arbitrary XML files to their equivalent representation in Prolog. The focus of the current development is, therefore, on postprocessing a rather general Prolog term tree representation of the Pillow library output into a more structured representation which will serve as the input to the Prolog analyser.

The Prolog B-Kernel itself is being written in SICStus Prolog with co-routining and constraints. The co-routining facilities are used to implement support for sophisticated data-structures such as sets, functions, relations, ... Below is part of the code of the B-Kernel, which hints at the fact that, thanks to the co-routining, the implementation was relatively straightforward.[5]

```
invert_relation(R,IR) :- when(nonvar(R),invert_relation2(R,IR)).
```

[5] Observe that, due to the lack of a sound negation, we have to write negated versions of predicates explicitly.

```
invert_relation2([],[]).
invert_relation2([(X,Y)|T],[(Y,X)|IT]) :-
     when(nonvar(T),invert_relation2(T,IT)).

partial_function(R) :- when(nonvar(R),pf(R,[])).
pf([],_).
pf([(X,_Y)|T],SoFar) :- not_element_of(X,SoFar), consistent(SoFar),
                        when(nonvar(T), pf(T,[X|SoFar])).

image(R,S,Res) :-  /*  Res = R[S] */
    when(nonvar(R), image2(R,S,Res)).
image2([],_,[]).
image2([(X,Y)|T],S,[Y|IT]) :- element_of(X,S), when(nonvar(T),image2(T,S,IT)).
image2([(X,_)|T],S,IT) :- not_element_of(X,S), when(nonvar(T),image2(T,S,IT)).
```

The constraint part is used to model (and later model check) the elementary data-structures. CLP over finite domains [5] is used to represent enumeration data types. In the future we also plan to use CLP(R) and CLP(Q) to represent arithmetic over reals and rationals.

Based on the technology presented in the earlier sections, we were able to develop a functional B-Animator with relatively little effort. A noteworthy feature of the animator is its ability to perform *symbolic animation* as well as concrete, ordinary animation. This allows a user to trace a B-machine symbolically, without providing actual values for the parameters (see Figure 5); the animator will set up constraints which the parameters have to satisfy (and checks whether concrete values exist which satisfy the constraints). This enables a symbolic exploration of the state space, but the user can at any time force the animator to provide concrete values.

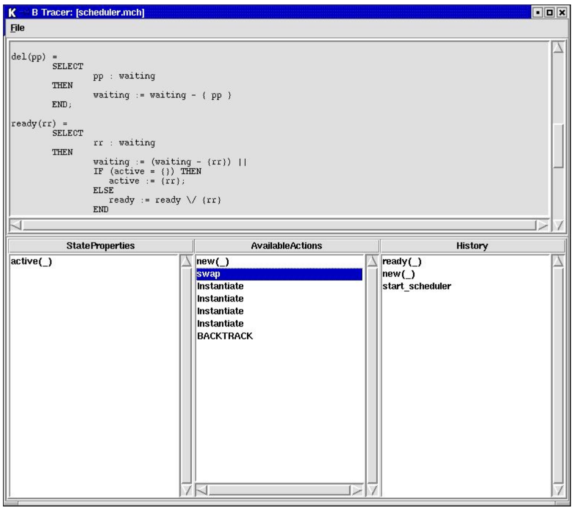

Fig. 5. Symbolic Animation of the Scheduler Example

We are also planning to provide temporal model checking [6] of CTL formulas, much in the same way as for CSP(LP) in Section 3. This allows one to check whether a given B-machine's temporal behaviour is as expected. Another kind of model checking [14], which we might call *state-based*, is more concerned about properties of the states and the preservation of invariants. Such a model checking is complementary to the temporal one and is especially useful in a B-setting [19]. We plan to achieve this model checking using constraint logic programming (and have already successfully applied it to small examples). The idea is to set up constraints which assert that the B-machine under consideration satisfies the invariant. We then apply an operation and set up constraints which assert that the invariant is no longer satisfied. If we manage to find concrete values which satisfy these constraints, we have uncovered a counter example to the B-machine's specification. A user could run such a model checker before attempting to prove a certain B-machine correct using existing B-tools. If our model checking tool uncovers a counter example it is clear that proving it will be in vain. Not only have we spared the user a lot of effort, the user also has a counter example at his or her disposal which will hopefully make it easier to correct the B-machine's specification. Eventually we also want to be able to verify correctness of B-refinement steps, by running Prolog verifier on proof obligations generated by B tool.

6 Conclusion

In this paper we have tried to demonstrate how Prolog technology can help us develop (and adapt) animators and verifiers for specification languages. We have highlighted how one can obtain specialised model checkers using partial evaluation technology. We have also discussed the possibility of using constraint logic programming to do symbolic animation and model-based model checking.

Our experience has shown that the latest generation Prolog systems are very well suited to prototype and develop specification languages. The power conferred by co-routining, constraints, and tabling make the promise of declarative programming much more of a reality than in a classical Prolog environment. Co-routining, constraints and tabling also obviate the need for a lot of extra-declarative features. This makes it often possible (e.g., in all of the applications described in this paper) to write a purely declarative kernel, which can then be fed into more powerful program analysis and optimisation tools.

All the work reported in this paper is work in progress, and it of course still remains to be seen how efficient and useful the developed tools will be. Nonetheless, we believe that Prolog's features have now reached a critical mass, enabling a whole new range of applications, which would take considerably more time to develop in other programming languages. We do not advocate Prolog to be used for all aspects of programming, and the latest Java-to-Prolog or C-to-Prolog interfaces actually allow an application to be partitioned into parts better suited for Prolog and parts better suited for imperative languages (e.g., the graphical user interface).

In terms of future directions for logic programming, we think that the actual performance of Prolog is nowadays very good (especially when compared to Java or when a partial evaluator can be used) and not an inhibiting factor. New developments, which we think would further increase the applicability and usefulness of Prolog systems in general and verification in particular are

– the integration of co-routining, constraints, and tabling into a single Prolog system;
– enabling constraint propagation between different constraint solvers and domains;
– a sound negation which properly handles co-routining (e.g., does not discard suspended goals in the negated call) and constraints;
– a standard type and module system, to ease the development of larger applications.

Acknowledgements

We would like to thank Bart Demoen, Michael Goldsmith, Ranko Lazic, Bill Roscoe, C.R. Ramakrishnan, Terrance Swift, and David S. Warren for their extremely useful comments, feedback, and explanations about CSP and XSB-Prolog respectively. I am also thankful to anonymous referees and Mauricio Varea for their feedback.

References

1. J.-R. Abrial. *The B-Book.* Cambridge University Press, 1996.
2. B-Core (UK) Limited, Oxon, UK. *B-Toolkit, On-line manual.*, 1999. Available at http://www.b-core.com/ONLINEDOC/Contents.html.
3. M. Butler. csp2B: A practical approach to combining CSP and B. *Formal Aspects of Computing*, 2000. To appear.
4. D. Cabeza and M. Hermenegildo. *The PiLLoW Web Programming Library.* The CLIP Group, School of Computer Science, Technical University of Madrid, 2001. Available at http://www.clip.dia.fi.upm.es/.
5. 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.
6. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 1999.
7. 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.
8. Declarative Systems and Software Engineering Group, Department of Electronics & Computer Science, University of Southampton, UK. *JavaLIL project.*, 1997. Available at http://www.ecs.soton.ac.uk/~amg/javalil/.
9. G. Delzanno and A. Podelski. Model checking in CLP. In R. Cleaveland, editor, *Proceedings of TACAS'99, the Second International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *LNCS*, pages 223–239. Springer-Verlag, 1999.

10. C. Ferreira and M. Butler. A process compensation language. In W. Grieskamp T. Santen B. Stoddart, editors, *Proceedings IFM'2000*. LNCS 1945, Springer-Verlag, 2000.
11. Formal Systems (Europe) Ltd. *Failures-Divergence Refinement — FDR2 User Manual*.
12. P. Henderson. Modelling architectures for dynamic systems. Technical report, University of Southampton, December 1999. To appear in LCNS. Available at http://www.ecs.soton.ac.uk/~ph/arc.htm.
13. C. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
14. D. Jackson, I. Schechter, and I. Shlyakhter. Alcoa: the alloy constraint analyzer. In *Proc. International Conference on Software Engineering*, Limerick, Ireland, June 2000.
15. 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.
16. M. Leuschel and J. Jørgensen. Efficient specialisation in Prolog using a hand-written compiler generator. Technical Report DSSE-TR-99-6, Department of Electronics and Computer Science, University of Southampton, September 1999.
17. 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.
18. M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialisation. In A. Bossi, editor, Logic-Based Program Synthesis and Transformation. *Proceedings of LOPSTR'99*, LNCS 1817, pages 63–82, Venice, Italy, September 1999.
19. L. Mikhailov and M. Butler. Combining b and alloy. Technical Report DSSE-TR-2001-2, Department of Electronics and Computer Science, University of Southampton, UK, February 2001.
20. 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.
21. 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 of the International Conference on Computer-Aided Verification (CAV'97)*, LNCS 1254, pages 143–154. Springer-Verlag, 1997.
22. A. W. Roscoe. Modelling and verifying key-exchange protocols using csp and fdr. In *IEEE Symposium on Foundations of Secure Systems*, 1995.
23. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1999.
24. J. B. Scattergood. *Tools for CSP and Timed-CSP*. PhD thesis, Oxford University, 1997.
25. Steria. *Atelier B, User and Reference Manuals*, 1996. Steria, Aix-en-Provence, France. Available at http://www.atelierb.societe.com/index_uk.html.
26. Tatibouet, Bruno. *The JBTools Package*, 2001. Available at http://lifc.univ-fcomte.fr/PEOPLE/tatibouet/JBTOOLS/BParser_en.html.
27. WebGain, Inc, 5425 Stevens Creek Blvd. Santa Clara, CA 95051, USA. *JavaCC - The Java Parser Generator*, 2001. Available at http://www.webgain.com/products/metamata/java_doc.html.

A Translation from B to Prolog

Small part of a B model to specify a safe Railway network.

```
    MACHINE Railway0
/*  B Model of Railway network based on requirements document from MTI
    Michael Butler, 13 Dec 2000 */

SETS
    TRAIN;
    CDV        /* Track sections */

ABSTRACT_VARIABLES
    aig_cdv,   /* Switches */
    net,       /* Total connectivity of sections */
    next,      /* Currrent connectivity based on switch positions */
    trns,      /* Set of trains on network */
    occp,      /* Occupancy function for sections */
    cdvA,      /* Initial cdv occupied by train */
    cdvZ       /* Final   cdv occupied by train */
. . .

LeaveCdvZ(t1,c1)  =
/*
Stop occupying cdvZ, i.e., change from
    ... -> 0 -> t1 -> ... -> t1 -> t1 -> 0 -> ...
to
    ... -> 0 -> t1 -> ... -> t1 -> 0  -> 0 -> ...
This is only possible if t1 also occupies next~(cdvZ).
*/
    PRE t1:TRAIN & c1:CDV THEN
    SELECT
        t1 : trns  &
        c1 = cdvZ(t1)  &
        c1 : dom(next~)  &
        occp(next~(c1)) = t1
    THEN
        cdvZ(t1) := next~(c1) ||
        occp := {c1} <<| occp
    END
    END;
. . .
END
```

Below is the corresponding part of the translation of this B-specification into Prolog, calling the B-Kernel which uses CLP(FD). This translation was still done by hand, but the automatic parser/compiler for AMN should soon be operational. Figure 6 shows the animator running this example.

```
/* Railway0.mch */
/* Prolog translation by Michael Leuschel, Feb 2001 */
. . .
type(_,any).
type(fd(X),train) :- X in 1..3.
type(fd(X),cdv) :- X in 1..6.
. . .
operation(leaveCdvZ(T1,C1),
          tuple(Aig_cdv,Net,Next,Trns,Occp,CdvA,CdvZ),
          tuple(Aig_cdv,Net,Next,Trns,Occp2,CdvA,CdvZ2)) :-
      /* PRE */
     type(T1,train),
     type(C1,cdv),
         /* SELECT */
     element_of(T1,Trns),
     apply_to(CdvZ,T1,C1),
```

```
    invert_relation(Next,InvNext),
    in_domain(C1,InvNext), /* c1 : dom(next~) */
    apply_to(InvNext,C1,Res),
    apply_to(Occp,Res,T1),  /* occp(next~(c1)) = t1  */
    /* THEN */
    override(CdvZ,T1,Res,CdvZ2), /* cdvZ(t1) := next~(c1) */
    domain_subtraction([C1],Occp,Occp2). /* occp := {c1} <<| occp */
...
```

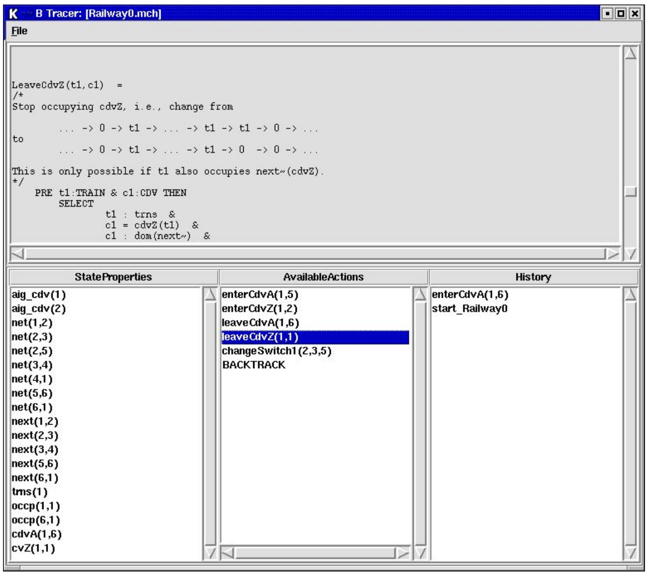

Fig. 6. Animation of the Railway Example