Logic Programming and Partial Deduction for the Verification of Reactive Systems: An Experimental Evaluation
(Abstract of ongoing work)

Michael Leuschel[1] and Thierry Massart[2]

[1] Department of Electronics and Computer Science
University of Southampton, e-mail: mal@ecs.soton.ac.uk
[2] Département d'Informatique
University of Brussels (ULB), e-mail: tmassart@ulb.ac.be

Abstract. In earlier work it has been shown that finite state CTL model checking of reactive systems can be achieved by a relatively simple interpreter written in tabled logic programming. This approach is flexible in the sense that various specification formalisms can be easily targeted (e.g., Petri nets, CSP, ...). Moreover, infinite state CTL model checking can be performed by analysing this interpreter using a combination of partial deduction and abstract interpretation. It has also been shown that this approach is powerful enough to decide coverability properties of various kinds of Petri nets.
In this ongoing work, we are empirically evaluating these approaches on various case studies of finite, parameterised and infinite systems. For finite state systems, we show how our approach and tool compares to standard tools for finite state model checking For parameterised or infinite state model checking, we are comparing our results with, e.g., XMC, Hytech.

1 Introduction

Recently there has been a lot of interest in applying logic programming techniques to model checking. Table-based logic programming and set-based analysis can be used as an efficient means of performing explicit model checking [RRR$^+$97][CP98]. Due to the built-in support of logic programming for non-determinism and unification, its also has a high potential both as a specification language and a language to build verification prototypes. If the system to verify is finite state, tabled logic programming environment can be used to automatically do the model-checking.

However, finite state model checking is sometimes not sufficient, since most *software* cannot be modelled directly by a *finite* state system. Moreover, distributed algorithms often have an unbounded number of instances. For these reasons, there has recently been considerable interest in *parameterised* and *infinite model checking.*

CTL Model Checking using Tabulation

A tabled logic programming systems such as XSB [SSW94] provides very efficient datastructures and algorithms to tabulate calls, i.e., it remembers which calls it has already encountered. This not only improves efficiency by avoiding the recomputation of already computed results, it also enables one to write logic programs in a more declarative style than using a "classical" Prolog system.

As was realised in [RRR+97] this enables one to use XSB as a basis to write very efficient model checkers, with relatively little effort. This has lead to the development of the XMC model checking system, whose performance is comparable to that of SPIN (at least for certain examples).

We have shown in [LM99] that, through a translation of CTL [EE81] into mu-calculus [Koz83] formulae using least and greatest fixpoints and into a further translation of the greatest fixpoints into least one's, the whole of CTL can be encoded as a relatively simple tabled logic program. Contrary to [RRR+97] our aim was not maximum efficiency, but writing a provably correct interpreter that can be fed into existing tools for the analysis and optimisation of logic programs.[1] One of the motivations is to use these analysis tools to perform infinite state model checking, as detailed below. Also, our CTL interpreter is independent of any underlying formalism. It only supposes that the successors of a state s can be computed (through a predicate `trans`) and that the elementary proposition of any state s can be determined (through a predicate `prop`). The interpreter can thus be easily applied to many formalisms, by providing appropriate encodings of `trans` and `prop`. An implementation of CTL as a (tabled) XSB Prolog program is given in Figure 1.

In this work, we examine to what extent this very simple CTL interpreter can be used on its own for finite state model checking. Our first experiments seem to show that the efficiency is surprsingly good, for such a simple system.

For example, we managed to verify the mutual exclusion of the Reader-Writer example ([ABC+95] resp. p154 and p.17) given by the Petri net (with an inhibitor arc) in Figure 2. This example models a system with K processes (tokens initially in place `P1`) which may request to read or write some file. To verify mutual exclusion properties we can simply run our CTL interpreter together with the encoding in Figure 3 (with `K` being instantiated various concrete values) in XSB-Prolog.

The Figure 4 contains preliminary results for this example as well as two other examples: a Central Server Model (CSM) and a Flexible Manufacturing System (FMS) [CM97]. XTL refers to our CTL interpreter running on XSB-Prolog 2.4. For the experiments we used a Macintosh Powerbook G3 with 320MB of RAM, running at 300Mhz using OS X 10.1.2. Compared to other timings published in the literature, our tools perform quite well, especially since they have not been designed with efficiency in mind.

We also did a few preliminary tests using the LOGEN partial evaluation system [JL96,LJ99] to specialise XTL for the particular system and formula at hand.

[1] These tools work best on declarative programs, and hence the full XMC system is probably not as amenable to analysis and optimisation by most existing tools.

```
/* A Model Checker for CTL fomulas written for XSB-Prolog */
:- table sat/2.
sat(_E,true).
sat(_E,false) :- fail.
sat(E,p(P)) :- prop(E,P). /* elementary proposition */
sat(E,and(F,G)) :- sat(E,F), sat(E,G).
sat(E,or(F,_G)) :- sat(E,F).
sat(E,or(_F,G)) :- sat(E,G).
sat(E,not(F)) :- tnot(sat(E,F)).
sat(E,en(F)) :- trans(_Act,E,E2),sat(E2,F). /* exists next */
sat(E,an(F)) :- tnot(sat(E,en(not(F)))). /* always next */
sat(E,eu(F,G)) :- sat_eu(E,F,G). /* exists until */
sat(E,au(F,G)) :- sat(E,not(eu(not(G),and(not(F),not(G))))),
                  sat_noteg(E,not(G)). /* always until */
sat(E,ef(F)) :- sat(E,eu(true,F)).  /* exists future */
sat(E,af(F)) :- sat_noteg(E,not(F)). /* always future */
sat(E,eg(F)) :- tnot(sat_noteg(E,F)). /* exists global */
            /* we want gfp -> negate lfp of negation */
sat(E,ag(F)) :- sat(E,not(ef(not(F)))). /* always global */

:- table sat_eu/3. /* tabulation to compute least-fixed point using XSB */
/* exists until */
sat_eu(E,_F,G) :- sat(E,G).
sat_eu(E,F,G) :- sat(E,F), trans(_Act,E,E2), sat_eu(E2,F,G).

:- table sat_noteg/2. /* tabulation to compute least-fixed point */
sat_noteg(E,F) :- sat(E,not(F)).
sat_noteg(E,F) :- findall(E2,trans(_A,E,E2),Succs), sat_noteg2(Succs,F).
sat_noteg2([],_).
sat_noteg2([S1|T],F) :- sat_noteg(S1,F), sat_noteg2(T,F).
```

Fig. 1. CTL interpreter

E.g., for the CSM case study this improves the runtime from 0.40 s to 0.28 s for 8 processes, from 1.64 to 1.05 s for 12 processes and from 4.55 to 4.53 for 16 processes. We will discuss the effect of specialisation in more detail at the workshop. We also plan to present a full empirical study on more case studies and comparing with other existing systems (all running on the same machine). For example, further experiments of finite state model checking will be done using a simplified CSP interpreter based upon [Leu01] and the results will be compared to FDR [Ros99,For].[2]

[2] The interpreter of [Leu01] has to be simplified (no complicated synchronisations will be allowed) so that the use of co-routining is prevented, which is not supported by XSB.

Infinite state model-checking

The method presented above, will not work for infinite state or parameterised model-checking, as during the execution of the CTL interpreter infinitely many *different* call patterns will arise. To (partially) solve this undecidable problem, we have used existing techniques for the *automatic* control of *logic program specialisation* [Leu99].

Now, an important question when attempting infinite state model checking in practice is: How can one *automatically* obtain an abstraction which is finite, but still as precise as required? A partial solution to this problem can be obtained by using existing techniques for the *automatic* control of *logic program specialisation* [Leu99]. First successful steps in that direction have been taken in [GL99,LM99], using the tools LOGEN [JL96,LJ99] and ECCE [LDS96,LMDS98].

[LL00b] gave a first formal answer about the power of the approach and showed that when we encode ordinary Petri nets as logic programs and use existing program specialisation algorithms, we can decide the so-called "coverability problems" (which encompass quasi-liveness, boundedness, determinism, regularity,...). This was achieved by showing that the Petri net algorithms by Karp–Miller [KM69] and Finkel [Fin93], which proceed forward, can be exactly mimicked. In [LL00a] we discuss how partial deduction can mimic backward algorithms as well. We prove that the backward algorithms scheme defined in [AČJT96,FS98,FS99] to solve the coverability problem of Well Founded Transition Systems as e.g. *reset Petri nets*, can also be mimicked by our environment.

The question we want to answer in this work is the practical performance of this approach. We have already applied our tools to the earlier mentioned parametric examples: a Central Server Model (CSM), a Reader-Writer model and a Flexible Manufacturing System (FMS). However, this time we have proven the safety of these models for *any* value of the parameter, and we have done so fully automatically. As can be seen in Figure 4 (in the ECCE column) these timings are again quite good, especially for a system that has initially been developed for another purpose. One can also see that already for relatively small values of the parameter, infinite state model checking is more efficient than the (incomplete) finite state model checking.

Also note that we managed to apply it to an example (Reader-Writer) with an inhibitor arc, for which Karp–Miller [KM69] and Finkel [Fin93] are not applicable. Indeed, the advantage of our approach is that it is always applicable, as long as we can provide a suitable logic programming encoding (however, we will not always obtain a decision procedure).

At the workshop we plan to compare our tools (on more examples) with, e.g., XMC and Hytech [HH95].

References

[ABC$^+$95] Marco Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets.* John Wiley & Sons, 1995.

[AČJT96] Parosh Aziz Abdulla, Kārlis Čerāns, Bengt Jonsson, and Yih-Kuen Tsay. General decidability theorems for infinite-state systems. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science*, pages 313–321, New Brunswick, New Jersey, 27–30 July 1996. IEEE Computer Society Press.

[CM97] Gianfranco Ciardo and Andrew S. Miner. Storage alternatives for large structured state spaces. In *Computer Performance Evaluation*, volume 1245 of *Lecture Notes in Computer Science*. Springer, 1997.

[CP98] Witolds Charatonik and Andreas Podelski. Set-based analysis of reactive infinite-state systems. In B. Steffen, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 1384, pages 358–375. Springer-Verlag, March 1998.

[EE81] E.M. Clarke and E.A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In D. Kozen, editor, *Proceedings of the Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71, Yorktown Heights, New York, May 1981. Springer-Verlag.

[Fin93] A. Finkel. The minimal coverability graph for Petri nets. *Lecture Notes in Computer Science*, 674:210–243, 1993.

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

[FS98] A. Finkel and P. Schnoebelen. Fundamental structures in well-structured infinite transition systems. In *Proceedings of LATIN'98*, LNCS 1380, pages 102–118. Springer-Verlag, 1998.

[FS99] A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere ! *Theoretical Computer Science*, 1999. To appear.

[GL99] Robert Glück and Michael Leuschel. Abstraction-based partial deduction for solving inverse problems – a transformational approach to software verification. In *Proceedings of the Third International Ershov Conference on Perspectives of System Informatics*, LNCS 1755, pages 93–100, Novosibirsk, Russia, 1999. Springer-Verlag.

[HH95] T. A. Henzinger and P.-H. Ho. HYTECH: The Cornell HYbrid TECHnology tool. *Hybrid Systems II*, LNCS 999:265–293, 1995.

[JL96] J. Jørgensen and M. Leuschel. Efficiently generating efficient generating extensions in Prolog. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation, International Seminar*, LNCS 1110, pages 238–262, Schloß Dagstuhl, 1996. Springer-Verlag.

[KM69] R. M. Karp and R. E. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 3:147–195, 1969.

[Koz83] Dexter Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[LDS96] Michael Leuschel and Danny De Schreye. Logic program specialisation: How to be more specific. In H. Kuchen and S.D. Swierstra, editors, *Proceedings of the International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'96)*, LNCS 1140, pages 137–151, Aachen, Germany, September 1996. Springer-Verlag.

[Leu99] Michael Leuschel. Logic program specialisation. In John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann, editors, *Partial Evaluation: Practice and Theory*, LNCS 1706, pages 155–188, Copenhagen, Denmark, 1999. Springer-Verlag.

[Leu01] Michael 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.

[LJ99] Michael Leuschel and Jesper 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.

[LL00a] Michael Leuschel and Helko Lehmann. Coverability of reset petri nets and other well- structuredtransition systems by partial deduction. In *Computational Logic - CL 2000, First International Conference, London, UK, 24-28 July, 2000, Proceedings*, volume 1861 of *Lecture Notes in Computer Science*. Springer, 2000.

[LL00b] Michael Leuschel and Helko Lehmann. Solving coverability problems of Petri nets by partial deduction. In Maurizio Gabbrielli and Frank Pfenning, editors, *Proceedings of PPDP'2000*, pages 268–279, Montreal, Canada, 2000. ACM Press.

[LM99] Michael Leuschel and Thierry Massart. Infinite state model checking by abstract interpretation and program specialisation. In Annalisa Bossi, editor, Logic-Based Program Synthesis and Transformation. *Proceedings of LOPSTR'99*, LNCS 1817, pages 63–82, Venice, Italy, September 1999.

[LMDS98] Michael Leuschel, Bern Martens, and Danny 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.

[Ros99] A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice-Hall, 1999.

[RRR$^+$97] Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, Scott A. Smolka, Terrance Swift, and David 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.

[SSW94] 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.

Fig. 2. Petri net of a Reader-Writer system with K processes

```
trans(t1,[s(X1),X2,X3,X4,X5,X6,X7],[X1,s(X2),X3,X4,X5,X6,X7]).
trans(t2,[X1,s(X2),X3,X4,X5,X6,X7],[X1,X2,s(X3),X4,X5,X6,X7]).
trans(t3,[X1,s(X2),X3,X4,X5,X6,X7],[X1,X2,X3,s(X4),X5,X6,X7]).
trans(t4,[X1,X2,s(X3),X4,s(X5),X6,X7],[X1,X2,X3,X4,s(X5),s(X6),X7]).
trans(t5,[X1,X2,X3,s(X4),s(X5),0,X7],[X1,X2,X3,X4,X5,0,s(X7)]).
trans(t6,[X1,X2,X3,X4,X5,s(X6),X7],[s(X1),X2,X3,X4,X5,X6,X7]).
trans(t7,[X1,X2,X3,X4,X5,X6,s(X7)],[s(X1),X2,X3,X4,s(X5),X6,X7]).
```

Fig. 3. Petri net example encode in logic programming

Case Study	Parameter Value	XTL	ECCE
CSM	2	0.01 s	-
	4	0.05 s	-
	6	0.20 s	-
	8	0.40 s	-
	12	1.64 s	-
	16	4.55 s	-
	32	55.03 s	-
	∞	-	4.44 s
FMS	1	0.03 s	-
	2	1.25 s	-
	3	71.69 s	-
	∞	-	55.10 s
Reader-Writer	2	0.00 s	-
	4	0.01 s	-
	6	0.04 s	-
	8	0.10 s	-
	12	0.49 s	-
	16	1.67 s	-
	32	55.96 s	-
	∞	-	1.49 s

Fig. 4. Preliminary results of our verification experiments