Infinite State Model Checking by Abstract Interpretation and Program Specialisation

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. We illustrate the use of logic programming techniques for finite model checking of CTL formulae. We present a technique for infinite state model checking of safety properties based upon logic program specialisation and analysis techniques. The power of the approach is illustrated on several examples. For that, the efficient tools LOGEN and ECCE are used. We discuss how this approach has to be extended to handle more complicated infinite state systems and to handle arbitrary CTL formulae.

1 Introduction

Recent years have seen dramatic growth [9] in the application of model checking [8, 5] techniques to the validation and verification of correctness properties of hardware, and more recently software systems. One of the methods is to model a hardware or software system as a finite, labeled transition system (LTS) which is then exhaustively explored to decide whether a given temporal specification holds. Recently, there has been increasing interest in applying logic programming techniques to model checking. Table-based logic programming can be used as an efficient means of performing explicit model checking [35] and set-based logic program analysis for model checking is explored in [7].

However, despite the success of model checking, most systems must be substantially simplified (i.e., *abstracted*) and "considerable human guidance and ingenuity is generally required to transform the original problem to a form where the final push button automation can be applied" [36]. Furthermore, most *software* systems can conveniently be modelled by infinite state systems: as soon as some kind of recursion, dynamic or unbounded data structures come into play, an unbounded number of states can be reached and must be verified. In practice the number of possible states in implementations is always finite but may be so huge as to make any exhaustive approach futile. This probably explains why, contrary to the situation in hardware, verification in general and model checking in particular has had hardly any impact on standard software practice.

For these reasons, there has recently been considerable interest in *infinite model checking* (e.g., [3, 39, 32, 12, 2]). This, by its generally undecidable nature, is a daunting task, for which *abstraction* is the key issue [9]. Indeed, abstraction allows one to approximate an infinite system (or a complicated finite one) by a (simple) finite one, and if proper care is taken the results obtained for the finite abstraction will be valid for the infinite system.

This research aims at exploring automatic means of building precise but tractable abstractions for infinite model checking (or model checking of finite, but very complex systems). We propose to do this by extending technology that has been developed to tackle similar problems in the context of automatic logic program analysis and specialisation. In essence, we:

– model a system to be verified as a logic program. This obviously includes finite LTS, but also allows to express systems with an infinite number of states. Note that this translations is often very straightforward, due to the built-in support of logic programming for non-determinism and unification.
– model the full temporal logic CTL as a logic program interpreter acting on the representation above. This interpreter will make use of the tight link that exists between the semantics of logic programs and least-fixed points.
– and then try to *automatically derive abstractions for infinite model checking through a combination of partial evaluation and abstract interpretation technology.* The tools LOGEN [19, 26] and ECCE [25, 27] are used successfully to automate the work.

The contribution of the paper is the development of a correct CTL interpreter in logic programming and its use as a sound basis for model checking of finite and infinite state systems via program analysis and specialisation in general and the tools LOGEN and ECCE in particular. This paper builds upon the initial insights and experiments in [14] where it was shown that abstraction-based partial deduction can be used as a powerful inversion tool.

In the following, we present the logic CTL [11] to express the properties we want to verify and its translation into a logic program. We discuss the validity of model checking infinite systems. We implement as logic programs, systems expressed as labeled transition systems, Petri nets, processes synchronised through shared variables, etc. We then show how existing technology for the specialisation and analysis of logic programs can be used to achieve some model checking tasks of infinite state systems. We present some successful experiments, but also shortcomings of the existing systems. We then give directions for further research to enable model checking of arbitrary CTL formulae on infinite state systems.

2 The model-checking of CTL in logic programming

2.1 CTL syntax and semantics

The temporal logic CTL (Computation Tree Logic) introduced by Clarke and Emerson in [11], allows to specify properties of specifications generally described as Kripke structures. The syntax and semantics for CTL are given below.

Given *Prop*, the set of *propositions*, the set of CTL formulae ϕ is inductively defined by the following grammar (where $p \in \mathit{Prop}$):

$$\phi := \mathit{true} \mid p \mid \neg\phi \mid \phi \wedge \phi \mid \forall \bigcirc \phi \mid \exists \bigcirc \phi \mid \forall \phi \mathcal{U} \phi \mid \exists \phi \mathcal{U} \phi$$

A CTL formula ϕ can be either true or false in a given state. For example, *true* is true in all states, $\neg$*true* is false in all states, and p is true in all states which contain the elementary proposition p. The symbol $\bigcirc$ is the *nexttime* operator and $\mathcal{U}$ stands for *until*. $\forall \bigcirc \phi$ (resp. $\exists \bigcirc \phi$) intuitively means that ϕ holds in every (resp. some) immediate successor of the current program state. The formula $\forall \phi_1 \mathcal{U} \phi_2$ (resp. $\exists \phi_1 \mathcal{U} \phi_2$) intuitively means that for every (resp. some) computation path, there exists an initial prefix of the path such that ϕ_2 holds at the last state of the prefix and ϕ_1 holds at all other states along the prefix.

The semantics of CTL formulae is defined with respect to a Kripke structure (S, R, μ, s_0) with S being the set of states, $R(\subseteq S \times S)$ the transition relation, $\mu(S \rightarrow 2^{Prop})$ giving the propositions which are true in each state, and s_0 being the initial state. Figure 1a gives a graphical representation of a Kripke structure with 3 states s_0, s_1, s_2, where s_0 is the initial state. The propositions p, q, r "label" the states. Generally it is required that any state has at least one

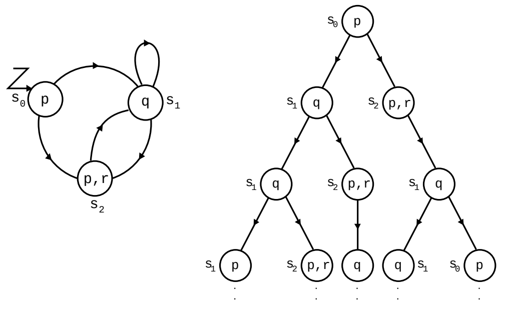

Fig. 1. Example of a Kripke structure.

outgoing vertex. From a Kripke structure, we can define an infinite transition tree as follows: the root of the tree is labelled by s_0. Any vertex labelled by s has one son labelled by s' for each vertex $s'$with a transition $s \rightarrow s'$ in the Kripke structure. For instance, the Kripke structure of Fig. 1a gives the prefix for the transition tree starting from s_0 given in Fig. 1b.

If the system is not directly specified by a Kripke structure but e.g. by a Petri net, the markings and transitions between them give resp. the states and transition relation of the Kripke structure.

The CTL semantics is defined on states s by:

- $s \models true$
- $s \models p$ *iff* $p \in P(s)$
- $s \models \neg\phi$ *iff* $\texttt{not}(s \models \phi)$
- $s \models \phi_1 \wedge \phi_2$ *iff* $s \models \phi_1$ and $s \models \phi_2$
- $s \models \forall \bigcirc \phi$ *iff* for all state t such that $(s, t) \in R$, $t \models \phi$
- $s \models \exists \bigcirc \phi$ *iff* for some state t such that $(s, t) \in R$, $t \models \phi$
- $s \models \forall \phi_1 \mathcal{U} \phi_2$ *iff* for all path $(s_0, s_1, \ldots)$ of the system with $s = s_0, \exists i (i \geq 0 \wedge s_i \models \phi_2 \wedge \forall j (0 \leq j < i \rightarrow s_j \models \phi_1))$
- $s \models \exists \phi_1 \mathcal{U} \phi_2$*iff* it exists a path $(s_0, s_1, \ldots)$ with $s = s_0$, such that $\exists i (i \geq 0 \wedge s_i \models \phi_2 \wedge \forall j (0 \leq j < i \rightarrow s_j \models \phi_1))$

If we look at the unfolding of the Kripke structure, starting from s_0 (Fig. 1b), we can see, for instance, that

- $s_0 \models \exists \bigcirc p$ since there exists a successor of s_0 (i.e. s_2) where p is true ($p \in \mu(s_2)$),
- $s_0 \not\models \forall \bigcirc p$ since in some successor of s_0, p does not hold.
- $s_0 \models \forall p \mathcal{U} q$ since the paths from s_0 either go to s_1 where q is true or to s_2 and then directly to s_1. In both cases, the paths go through states where p is true before reaching one state where q holds.

Since they are often used, the following abbreviations are defined

- $\forall\Diamond\phi \equiv \forall \text{ true } \mathcal{U}\phi$ i.e. for all paths, ϕ eventually holds,
- $\exists\Diamond\phi \equiv \exists \text{ true } \mathcal{U}\phi$ i.e. there exists a path where ϕ eventually holds,
- $\exists\Box\phi \equiv \neg\forall\Diamond(\neg\phi)$ i.e. there exists a path where ϕ always holds,
- $\forall\Box\phi \equiv \neg\exists\Diamond(\neg\phi)$ i.e. for all paths ϕ always holds.

E.g. $\forall\Box\phi$ states that ϕ is an invariant of the system, $\forall\Diamond\phi$ states that ϕ is unavoidable and $\exists\Diamond\phi$ states that ϕ may occur.

2.2 CTL as fixed point and its translation to logic programming

One starting point of this paper is that both the Kripke structure defining the system under consideration and the CTL specification can be easily defined using logic programs. This is obvious for the standard logic operators as well as for the CTL operators $\exists\bigcirc$. The following equality can be easily proved $\forall \bigcirc \phi \equiv \neg\exists\bigcirc\neg\phi$. Moreover, the operators $\exists\mathcal{U}$ and $\forall\mathcal{U}$ can be defined as fixpoint solutions [31]. Indeed, if you take the view that ϕ represents the set of states S where ϕ is valid, it can be proved that $\exists p\mathcal{U}q \equiv \mu Y = (q \vee (p \wedge \exists \bigcirc Y))$ where μ stands for the least fixpoint of the equation. Intuitively this equation tells that the set of states satisfying $\exists p\mathcal{U}q$ is the least one satisfying q or having a successor which satisfies this property.

$\exists\Diamond$ and $\forall\Box$ can be derived from what precedes. $\forall\bigcirc$ can be derived using the equivalence $\forall \bigcirc \phi \equiv \neg\exists \bigcirc \neg\phi$. Slightly more involved is the definition of the set of states which satisfy $\exists\Box\phi$. These states can be expressed as the solution of $\nu X = \phi \wedge \exists \bigcirc X$ where ν stands for the greatest fixpoint of the equation. Greatest fixpoint cannot be directly computed by logic programming systems, but the equation can be translated into $\exists\Box\phi \equiv \neg\Phi$ where the set Y of states which satisfy Φ is defined by $\mu Y = \neg\phi \vee \neg\exists \bigcirc \neg Y$. Finally, $\forall\Diamond\phi \equiv \neg\exists\Box\neg\phi$ and $\forall\phi_1\mathcal{U}\phi_2 \equiv \neg(\exists\neg\phi_2\mathcal{U}(\neg\phi_1 \wedge \neg\phi_2)) \wedge \neg\exists\Box\neg\phi_2$. This last equality says that a state satisfies $\forall\phi_1\mathcal{U}\phi_2$ if it has no path where ϕ_2 is continuously false until a state where both ϕ_1 and ϕ_2 are false nor a path where ϕ_2 is continuously false.

We can see that we only use least fixpoint of monotonic equations. Moreover, the use of table-based Prolog (XSB Prolog) ensures proper handling of cycles.

For infinite state systems, the derivation of the fixpoints cannot generally be completed. However, by the monotonicity of all the used fixpoints, all derived implications belong indeed to the solution (no overapproximation).

Our CTL specification is independant of any model. It only supposes that the successors of a state s can be computed (through the predicate `trans`) and that the elementary proposition of any state s can be determined (through `prop`). In Fig. 2 we present a particular implementation of CTL as a (tabled) logic program. For example, it can be used to verify the mutual exclusion example from [8], simply by running it in XSB-Prolog. This follows similar lines as [35, 28] where tabled logic programming is used as an (efficient) means of *finite* model checking. Nonetheless, our translation of CTL in this paper is expressed (more clearly) as a meta-interpreter and will be the starting point for the model checking of *infinite* state systems using program specialisation and analysis techniques.

```
/* A Model Checker for CTL fomulas written for XSB-Prolog */
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)) :- not(sat(E,F)).
sat(E,en(F)) :- trans(_Act,E,E2),sat(E2,F). /* exists next */
sat(E,an(F)) :- not(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)) :- not(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. /* table to compute least-fixed point using XSB */
sat_eu(E,_F,G) :- sat(E,G).  /* exists until */
 sat_eu(E,F,G) :- sat(E,F), trans(_Act,E,E2), sat_eu(E2,F,G).
:- table sat_noteg/2. /* table to compute least-fixed point using XSB */
sat_noteg(E,F) :- sat(E,not(F)).
sat_noteg(E,F) :- not((trans(_Act,E,E2),not(sat_noteg(E2,F)))).
```

Fig. 2. CTL interpreter

Model checking of infinite systems The infinite state system we handle are finitely branching. In fact, what is required is to be able, in a finite number of "steps", to look at all the succesors of a state. (In real-time systems a state can have an infinite number of direct successors, and model checking then requires more sophisticated symbolic methods.) We have to show that, using these infinite state systems, our CTL interpreter is correct wrt. the SLS-semantics [34]. This is because our analysis and specialisation might replace infinite failure by finite failure but do so only in accordance with the SLS-semantics.

Indeed, the only potential loops are linked with $\exists\mathcal{U}$ or $\forall\mathcal{U}$. In fact, we can notice that the computation is based on two fixpoint calculations; one is defined through `sat_eu` and the other through `sat_noteg`. If we look at `sat_eu` (the treatment of `sat_noteg` will pose similar problems), three cases may occur in the SLD-resolution:

- A path is found satisfying F *Until* G (success; no difference with SLDNF-semantics).
- It is found that no path satisfies F *Until* G, i.e. all the paths satisfy $\neg G \mathcal{U} (\neg F \vee \neg G)$ (finite failure; no difference with SLDNF-semantics)
- The resolution loops in an infinite path satisfying F. In this case the system will not reply if no means is given to detect this infinite path. However, wrt the SLS-semantics the answer is *no* (which is different from the SLDNF-semantics), which is the correct answer according to the CTL-semantics we have given earlier (no path satisfies the requested property).

3 The systems analysed

We illustrate our approach by analysing finite or simple but infinite states systems specified initially by LTS, Petri nets or parallel processes using shared variables. This section presents these systems.

3.1 Petri net of a simple mutual exclusion problem

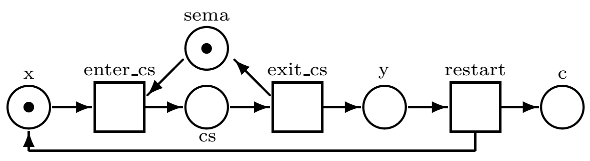

Fig. 3. Petri net with a single semaphore

Figure 3 models a single process which may enter a critical section (cs), the access to which is controlled by a semaphore (sema). This Petri net can be encoded directly as `trans/3` facts for our CTL interpreter:

```
trans(enter_cs,[s(X),s(Sema),CritSec,Y,C],[X,Sema,s(CritSec),Y,C]).
trans(exit_cs, [X,Sema,s(CritSec),Y,C],[X,s(Sema),CritSec,s(Y),C]).
trans(restart,[X,Sema,CritSec,s(Y),C],[s(X),Sema,CritSec,Y,s(C)]).
```

3.2 Petri net of a manufacturing system

The manufacturing system in Fig. 4, used in [3], has been analysed. It models an automated manufacturing system with 4 machines, 2 robots, 2 buffers (x_{10} and x_{15}) and an assembly cell. The initial marking is such that $x_1 = p$ for some nonnegative parameter p. In [3], Bérard and Fribourg have used Hytech [17] to discover a potential deadlock when $p > 8$.

3.3 Producer/consumer processes with shared variables

The following is a simplified version of the producer/consumer example from [1], using a buffer of length 1. A state of the system is not only, as in the previous cases, a tuple of natural numbers but this time contains items (simple values) and lists of items. A "state" of the system is given by `[A,Prod,In,Out,Buf,CR,B]` where `A` is the list of items which remain to be produced, `Prod` the last item which has been produced, `In` the number of empty places in the buffer, `Out` the number of full places in the buffer, `Buff` the buffer and `CR` the last item which has been removed by the consumer. Finally `B` allows to check the items which remain to be consumed (and is initially a copy of `A`).

```
trans(prod,[[X|T],prod,In,Out,Buf,CR,B], [T,add(X),In,Out,Buf,CR,B]).
trans(add, [A,add(X),1,0,Buf,CR,B], [A,prod,0,1,X,CR,B]).
trans(rem, [A,PA,0,1,Buf,cons,B], [A,PA,1,0,Buf,rem(Buf),B]).
trans(cons,[A,PA,In,Out,Buf,rem(Buf),[Buf|B]], [A,PA,In,Out,Buf,cons,B]).
trans(err ,[A,PA,In,Out,Buf,rem(Buf),[B2|B]],
          [err,err,err,err,err,err,err]) :- Buf\==B2.
```

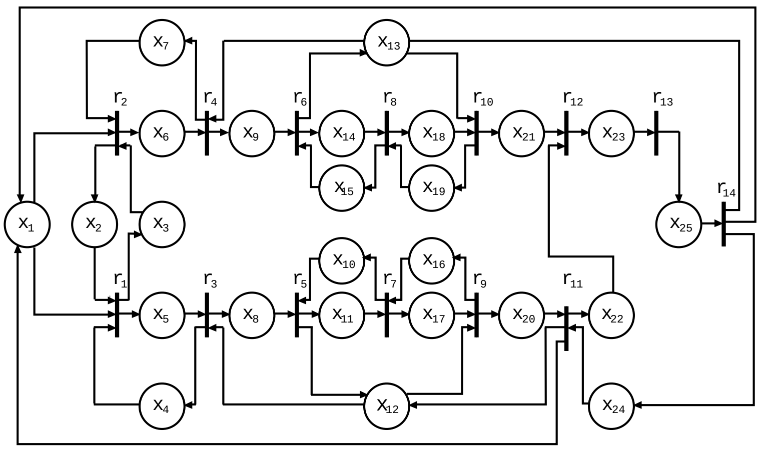

Fig. 4. Petri net representation of an automated manufacturing system with four machines, two robots, two buffers (x_{10}, x_{15}) and an assembly cell.

4 Infinite Model Checking of Safety Properties

Before attempting to verify any CTL formula, let us restrict our attention to checking safety properties, i.e., formulae of the form $s \models \forall\Box safe$. Model checking of such properties amounts to showing that there exists no trace which leads to an invalid state, i.e., exploiting the fact that $\forall\Box safe \equiv \neg\exists\Diamond(\neg safe)$.

Consider the Petri net in subsection 3.1. We first have to specify properties of interest by defining `prop/2`: `prop([X,Sema,s(s(CritSec)),Y,C],unsafe)`.

Here, we have specified that a state is unsafe if two or more processes are in their critical section at the same time. Now, to check whether the above Petri net can reach an unsafe state for an initial marking with 1 token in the semaphore (sema), 0 tokens in the reset counter (c), no processes in the critical section (cs), no processes in the final place (y) and `X` processes in the initial place (x) we simply run the query: `sat([X,0,s(0),0,0],ef(p(unsafe)))`.

Unfortunately, this query does not terminate under Prolog or XSB-Prolog although the system does indeed satisfy the safety property for any value of X. Even if we try to prove the property just for a particular value of X (e.g., $X = s(0)$) neither Prolog nor XSB-Prolog [37] will terminate (even when adding moding or delay declarations). Indeed, the queries have infinitely failed SLD/SLG-trees with an infinite number of distinct call patterns (due to the counter c). Thus, according to the well-founded semantics, we have that the program indeed entails `not(sat([X,0,s(0),0,0],ef(p(unsafe))))`, but existing procedures are unable to establish this.

In the following, we will be able to prove this safety property by a *semantics-preserving program specialisation and analysis technique*, specialising the CTL interpreter for the query `sat([X,0,s(0),0,0],ef(p(unsafe)))` to the empty program (in a logically sound fashion). For this we will proceed in three phases:

1. specialise the full CTL interpreter for the particular property using an offline specialiser, so as to get rid of unneccesary complexities.
2. specialise the so obtained simplified interpreter using a full-fledged online specialiser so as to obtain a finite, and as precise as possible abstraction of the infinite state system for the property at hand.
3. use abstract interpretation to analyse this finite representation and determine whether the property is true, false, or undecided.

In the following we will describe and illustrate each of these phases.

4.1 Pre-compilation with LOGEN

The complete CTL interpreter of Fig. 2 is quite complex, and makes heavy usage of negation. The latter is difficult to handle by most current program analysis and specialisation tools. However, for the particular class of formulae we want to handle in this section we can get rid of the unnecessary complexity of full CTL by pre-compiling the interpreter for the formula to be checked. This task is best performed by an offline specialiser, such as the LOGEN system [19, 26]. For instance, specialising Fig. 2 for the query `sat([Processes,0,s(0),0,0],ef(p(unsafe)))` yields the much simplified interpreter below, which contains no negation and where the Petri net has been compiled into the interpreter:

```
/* benchmark info: 1.17 ms */
/* atom specialised: sat([_264,0,s(0),0,0],ef(p(unsafe))) */
sat__0(B) :- sat_eu__1(B).
sat_eu__1([B,C,s(s(D)),E,F]).
sat_eu__1([s(G),s(H),I,J,K]) :- sat_eu__1([G,H,s(I),J,K]).
sat_eu__1([L,M,s(N),O,P]) :- sat_eu__1([L,s(M),N,s(O),P]).
sat_eu__1([Q,R,S,s(T),U]) :- sat_eu__1([s(Q),R,S,T,s(U)]).
```

This interpreter should now be called using the predicate `sat__0(P)` (which corresponds to calling `sat([P,0,s(0),0,0],ef(p(unsafe)))` in the original program). Observe that LOGEN (and ECCE as well) concatenates two underscores and a unique identifier to existing predicate names.

The exact details of this compilation phase are not relevant for the present article; all we need to know is that it terminates (actually it is also very efficient) and that it is totally correct in the sense that it preserves the computed answers and finite failure. In other words, the specialised program succeeds for a specialised query (e.g., `sat__0(s(X))`) with a computed answer θ (respectively finitely fails) iff the original program does so for the corresponding original query (e.g., `sat([s(X),0,s(0),0,0],ef(p(unsafe)))`.

Observe that even this much simplified interpreter still does not terminate under either SLD or tabled SLG resolution. Below, we will show how this interpreter can be analysed using automatic techniques and how we can show that the encoded, infinite state Petri net obeys the safety property.

4.2 Online partial deduction with ECCE

The second phase of our model checking technique will perform online partial deduction using the ECCE tool. It will construct a finite representation of the infinite state space in the form of a specialised program. Below we present the essential details of partial deduction and the essential details of the algorithm used by ECCE.

The underlying technique of partial deduction is to construct finite but possibly incomplete SLD(NF)-trees (i.e. a SLD(NF)-tree which, in addition to success and failure leaves, may also contain leaves where no literal has been selected for a further derivation step). These incomplete SLD(NF)-trees are obtained by applying an unfolding rule, defined as follows.

Definition 1. *An* unfolding rule *is a function which, given a program P and a goal G, returns a finite non-trivial*[1] *and possibly incomplete SLD(NF)-tree τ for $P \cup \{G\}$. We also define leaves(τ) to be the atoms in the leaf goals of τ.*

Formally, the resultant of a branch of τ leading from the root G to a leaf goal G_i via computed answer θ is the formula $G\theta \leftarrow G_i$. Partial deduction uses the resultants for a given set of atoms $\mathcal{S}$ to construct the specialised program (and for each atom in $\mathcal{S}$ a different specialised predicate definition will be generated). Under the conditions stated in [29], namely *closedness* (all leaves are an instance of an atom in $\mathcal{S}$) and *independence* (no two atoms in $\mathcal{S}$ have a common instance), total correctness of the specialised program is guaranteed (i.e., as above we preserve the computed answers and finite failure).

We now present a concrete partial deduction algorithm based upon [27]. This algorithm structures the set S of atoms to be specialised as a *global tree* γ: i.e., a tree whose nodes are labeled by atoms and where A is a descendant of B if specialising *label*(B) lead to the specialisation of *label*(A). It outputs a set of atoms $\mathcal{A}$ which can be used to construct a totally correct specialisation of P

[1] A trivial SLD(NF)-tree is one in which no literal in the root has been selected for resolution. Such trees are disallowed to obtain correct partial deductions.

for all instances of $\leftarrow Q$ (possibly using a renaming transformation to ensure independence).

Note that the algorithm below can be seen as a special kind of *forwards* abstract interpretation (see [24]), where each atom in γ actually denotes all its instances (i.e., the concretisations $\gamma(A)$ of an atom A are all the instances of A).

To ensure termination of our algorithm, we have to ensure that the unfolding rule U builds a finite SLD(NF)-tree. In addition, we have to guarantee that no infinite branches are built up in the global tree γ: If it looks like an infinite branch is being built up we have to abstract some of the atoms and restart the process. We thus also have to ensure that this abstraction process itself cannot be repeated infinitely often.

The following auxiliary concepts will help us to achieve this feat. First, to be able to perform a suitable generalisation we define:

Definition 2. *The* most specific generalisation *of a finite set of expressions S, also denoted by $msg(S)$, is the most specific expression M such that all expressions in S are instances of M.*

Algorithms for calculating the *msg* exist [21], and we have for example $msg(\{p(0, s(0)),\ p(0, s(s(0)))\}) = p(0, s(X))$. We also have the important property, that for every expression A, there are no infinite chains of strictly more general expressions. Now, to detect infinite branches, both in the global tree γ and the SLD(NF)-trees constructed by U, we will use the homeomorphic embedding relation derived from [18, 20]. The following is the definition from [38]:

Definition 3. *The* (pure) homeomorphic embedding *relation $\trianglelefteq$ on expressions is inductively defined as follows (i.e. $\trianglelefteq$ is the least relation satisfying the rules):*

1. *$X \trianglelefteq Y$ for all variables X, Y*
2. *$s \trianglelefteq f(t_1, \ldots, t_n)$ if $s \trianglelefteq t_i$ for some i*
3. *$f(s_1, \ldots, s_n) \trianglelefteq f(t_1, \ldots, t_n)$ if $\forall i \in \{1, \ldots, n\} : s_i \trianglelefteq t_i$.*

(Notice that n is allowed to be 0 and we thus have $c \trianglelefteq c$ for all constant and proposition symbols). The intuition behind the above definition is that $A \trianglelefteq B$ iff A can be obtained from B by "striking out" certain parts, or said another way, the structure of A reappears within B. We have the important property ([18, 20]) that $\trianglelefteq$ is a so-called well-quasi order on the set of expressions over a finite alphabet, i.e., for every infinite sequence $s_1, s_2, \ldots$ of expressions there exists $i < j$ such that $s_i \trianglelefteq s_j$.

Algorithm 4.1 (*partial deduction algorithm*)

```
Input: a program $P$ and a goal $\leftarrow Q$
Output: a set of atoms $\mathcal{A}$ and a global tree $\gamma$
Initialisation: $\gamma$ := a "global" tree with a single unmarked node, labelled by $Q$
repeat
  pick an unmarked leaf node $L$ in $\gamma$
  if $\exists$ a marked variantof $L$ in $\gamma$ then   mark $L$
  else if $\exists$ ancestor $W$ of $L$ such that $label(W) \trianglelefteq label(L)$ then
    $label(W) := msg(L, W)$
    remove all descendants of $W$ and unmark $W$
  else
    mark $L$
    for all $A \in leaves(U(P, label(L)))$ do
      add a new unmarked $C$ child of $L$ to $\gamma$
      $label(C) := A$
      $label(L \to C) :=$ a characteristic path i.e. a sequence of clauses in $P$ which were
    resolved with
until all nodes are marked
output $\mathcal{A} := \{label(A) \mid A \in \gamma\}$
```

In this algorithm, M is a variant of L iff for some θ_1, θ_2: $M\theta_1 = L$ and $L\theta_2 = M$. The notion of *characteristic path* is developped in [13, 27].

The above algorithm is parametrised by an unfolding rule U. Upon termination of the algorithm the closedness condition of [29] is satisfied, i.e., it is ensured that *together* the atoms $\mathcal{A}$ with their SLD(NF)-trees form a *complete description* of all possible computations that can occur for all concrete instances $\leftarrow A\theta$ of the goal of interest.

Note that ECCE can also handle entire conjunctions of atoms [10] (instead of just single atoms), but this was not required for the experiments in this paper. By default, its control is also more refined in that it uses an extended $\trianglelefteq$ [27, 22] and characteristic trees on top of syntactic structure to control abstraction [27].

4.3 Most Specific Version Abstract Interpretation

The task of the abstract interpretation phase will be to do the verification proper and try to infer whether the finite representation produced by the previous phase admits a solution. For this we will use an abstract interpretation method based on [30] which calculates so-called most specific versions of programs.

By $mgu^*(A, B)$ we denote a particular idempotent and relevant most general unifier of A and some B', obtained from B by renaming apart wrt A (i.e., so that B' and A have no variables in common). We also define the predicate-wise application msg^* of the msg: $msg^*(S) = \{msg(S_p) \mid p \in Pred(P)\}$, where S_p are all the atoms of S having p as predicate and $Pred(P)$ denotes the set of predicates occurring in the program P. In the following we define the well-known *non-ground* T_P operator (whose least fixed point gives the S-semantics [4]) along with an abstraction T_P^α of it.

Definition 4. *For a definite logic program P and a set of atoms $\mathcal{A}$ we define: $T_P(\mathcal{A}) = \{H\theta_1 \dots \theta_n \mid H \leftarrow B_1, \dots, B_n \in P \wedge \theta_i = mgu^*(B_i\theta_1 \dots \theta_{i-1}, A_i)$ with $A_i \in \mathcal{A}\}$. We also define $T_P^\alpha(\mathcal{A}) = msg^*(T_P(\mathcal{A}))$.*

One of the abstract interpretation methods of [30] can be seen as calculating $lfp(T_P^\alpha) = T_P^\alpha \uparrow^\infty (\emptyset)$ (this in turn can be seen as an abstract interpretation method which infers top level functors for every predicate). The idea is to initially proceed like T_P, but if we get two or more success patterns for the same predicate then we retain only one success pattern which covers them. $T_P^\alpha \uparrow^\infty (\emptyset)$ will always stabilise after a finite number of iterations and it will produce a safe approximation of the success set (i.e., any call which does not unify using mgu^* with an element of $T_P^\alpha \uparrow^\infty (\emptyset)$ will fail [finitely or infinitely]). In [30] more specific versions of clauses and programs are obtained in the following way (which preserves the least Herbrand model and the computed answers, but may replace infinite by finite failure):

Definition 5. *Let $C = H \leftarrow B_1, \ldots, B_n$ be a definite clause and $\mathcal{A}$ a set of atoms. We define: $msv_{\mathcal{A}}(C) = \{C\theta_1 \ldots \theta_n \mid \theta_i = mgu^*(B_i\theta_1 \ldots \theta_{i-1}, A_i)$ with $A_i \in \mathcal{A}\}$. The* more specific version *$msv(P)$ of a program P is then obtained by replacing every clause $C \in P$ by $msv_{lfp(T_P^\alpha)}(C)$ (note that $msv_{lfp(T_P^\alpha)}(C)$ contains at most 1 clause).*

Notably, the most specific version of a program without facts is the empty program! Also observe that the above described analysis works *backwards* (or bottom-up) from the facts to the query. It is thus an ideal complement to the forwards analysis that partial deduction performs. (Ideally one would like to perform the forwards and backwards analysis together [25]; generic algorithms for this exist [25, 24] but they are not yet implemented.)

4.4 Putting it all together

Let us now return to checking the earlier mentioned safety property $\neg\exists\Diamond(unsafe)$ of the Petri net of Fig. 3, meaning that it is *impossible* to reach a marking where two processes are in their critical section at the same time.

We have already compiled the formula $\exists\Diamond(unsafe)$ and our particular Petri net into the interpreter by LOGEN in Section 4.1.

Let us first attempt to prove that property for 2 processes. We thus apply ECCE to the compiled interpreter, specialising it for `sat__0([s(s(0)),0,s(0),0,0])` (i.e., an initial marking with 2 processes and 1 token in the semaphore). This yields the following specialised program (after a transformation time of 0.5 s):

```
sat__0([s(s(0)),0,s(0),0,0]) :- sat__0__1.
sat__0__1 :- sat_eu__1__2.
sat__0__1 :- sat_eu__1__3.
sat_eu__1__2 :- sat_eu__1__7(s(s(0)),s(0)).
sat_eu__1__2 :- sat_eu__1__3.
sat_eu__1__3 :- sat_eu__1__4.
sat_eu__1__4 :- sat_eu__1__5(s(s(0)),s(0)).
sat_eu__1__4 :- sat_eu__1__3.
sat_eu__1__5(A,s(B)) :- sat_eu__1__6(A,B).
sat_eu__1__5(s(A),B) :- sat_eu__1__5(A,s(B)).
sat_eu__1__6(A,B) :- sat_eu__1__5(s(A),B).
sat_eu__1__6(s(A),B) :- sat_eu__1__6(A,s(B)).
sat_eu__1__7(A,s(B)) :- sat_eu__1__8(A,B).
sat_eu__1__7(s(A),B) :- sat_eu__1__5(A,s(B)).
sat_eu__1__8(A,B) :- sat_eu__1__7(s(A),B).
sat_eu__1__8(s(A),B) :- sat_eu__1__8(A,s(B)).
```

As you can see ECCE always generates one clause (the first one, defining `sat__0`) which allows the specialised program to be used in the same way as the original one and then clauses for a renamed (two underscores and the number 1 are usually added to the predicate name) and filtered (only variables occuring in the query are left as arguments) version of the query (the clauses defining `sat__0__1`). As this program contains no facts, the most specific version transformation trivially produces:

```
    sat__0([s(s(0)),0,s(0),0,0]) :- fail.
```

This establishes the safety property: $\exists\Diamond(unsafe)$ is false and there is no way that the system can reach a state where `unsafe` holds. As already mentioned, this task cannot be established by PROLOG or XSB-PROLOG [37] with tabling.

Similarly, one can prove the safety property *regardless* of the number of processes, i.e., for *any* number of tokens in the initial place (x). When we specialise the same compiled version of Fig. 2 for the query `sat__0([X,0,s(0),0,0])` and then compute the most specific version we get the following (in similar transformation times):

```
    sat__0([X,0,s(0),0,0]) :- fail.
```

There are now of course two important questions that arise:

1. How can we be sure that the above implies the safety property? In fact, if the safety property of the system in Fig. 3 is not satisfied, there must be a trace of finite length leading to an unsafe state. Hence, by completeness of SLD (and correctness of the CTL interpreter) we can deduce that there should have been a computed answer for the query: `sat__0([X,0,s(0),0,0])`. Now, as LOGEN, ECCE, and the most specific program technique all preserve (provided there are no bugs in the implementations of course) failure and the computed answers (cf. [29] and [30] respectively), and as the specialised program fails we can conclude that the safety property does hold.
   Indeed, the specialised program produced by any of the 3 phases is totally correct: they do not remove any computed answer nor do they add any. So, in a sense there is no over-approximation or under-approximation! Approximations only come into play if we analyse the residual programs.
   For instance, we can produce the following *safe over-approximation* of the success set: deduce that a call $p(\bar{t})$ fails if it unifies with no clause in the residual program (or just with a single clause $p(\bar{s}) \leftarrow fail$). For all other calls deduce that they potentially succeed. Similarly, we can extract a *safe under-approximation* of the success set: deduce that a call $p(\bar{t})$ succeeds if there is a fact $p(\bar{s}) \leftarrow$ in the residual program such that $p(\bar{t})$ is an instance of $p(\bar{s})$. Otherwise deduce that the call potentially fails.
2. How did the system achieve this (automatically) ? In essence, the *specialisation component (*ECCE) performed a symbolic traversal of the infinite state space, thereby producing a finite representation of it, on which the analysis component performed the verification of the specification. More precisely, the following ingredients of our system seem to be relevant or even vital:
   – the homeomorphic embedding $\trianglelefteq$ ensures that we build a finite representation of the state space. At the same time $\trianglelefteq$ is sufficiently powerful [23]

to minimise unnecessary abstraction, increasing the chances of successful verification.
- characteristic trees ensure that we produce enough polyvariance to account for different behaviour of configurations. It further minimises the risk of unnecessary, harmful abstraction (cf. [27]).
- abstract interpretation performs the model checking proper on the finite representation obtained above.
- the abstract interpretation works backwards (from unsafe states towards initial states) while the partial deduction works forwards (from initial states to unsafe states). Our technique thus gives a combined backwards/forwards analysis (which should become even better by implementing the full integration of [25, 24]).

4.5 Tackling the manufacturing Petri net example

We applied the approach to the manufacturing system in subsection 3.2 and were able to prove absence of deadlocks for parameter values of e.g., 1,2,3. When leaving the parameter unspecified, the system did not establish an absence of deadlocks and produced a (large) residual program containing facts. And indeed, for parameter values ≥ 9 the system can actually deadlock. We are investigating whether a counter example can effectively be extracted from this residual program. The runtimes of our system compare favourably with HyTech [3].

5 Coping with more complicated formalisms

In principle, it is possible to extend our approach to verify larger, more complicated infinite systems. (Notice that larger systems have been approached with related techniques as a preprocessing phase [16]. However, their purpose is to reduce the state space rather than provide novel ways of reasoning.) As with all automatic specialisation tools, there are several points that need to be addressed: allow more generous unfolding and polyvariance (efficiency, both of the specialisation process and the specialised program, are less of an issue in model checking) to enable more precise residual programs and implement the full algorithm of [24] which allows for more fine grained abstraction and use BDD-like representations whenever possible. We elaborate on some of these issues below.

Also, in theory, we can apply the power of our approach, to systems specified in other formalisms such as the π-calculus (cf. the experiment in [15]) or processes with synchronisations, simply by writing an interpreter for these formalisms in logic programming.

Looking at the producer/consumer example presented in 3.3, we use the same interpreter for Petri nets as in Fig. 3 but for more complex states. An error occurs in the system of Subsection 3.3 when the next item consumed does not correspond to the one expected. To encode this as well as the possible inital states of our system, we add:

```
prop([err,err,err,err,err,err,err],unsafe).
err(A) :- sat([A,prod,1,0,vide,cons,A],ef(p(unsafe))).
```

Again, our 3-phased approach has achieved infinite model checking and has inferred `err(A) :- fail.`, i.e. for *any* list of items, the safety property is verified.

However, things are not always that easy and problems do appear with more complicated systems.

Example 1. The following is a slightly more involved version of the produce-consume example from [1]. It still uses a buffer of length 1, but uses arithmetic operations to test whether the buffer contains any item to consume. Here, `In` (resp. `Out`) stands for the number of items which have been put in (resp. removed from) the buffer. (In the full version of [1], we have tests of the form $In < Out + N$, where N is the size of the buffer.)

```
trans(prod,[N,[X|T],prod,In,Out,Buf,CR,B], [N,T,add(X),In,Out,Buf,CR,B]).
trans(add, [N,A,add(X),In,Out,Buf,CR,B],
           [N,A,prod,In1,Out,X,CR,B]) :- In < Out+1, In1 is In+1.
trans(rem, [N,A,PA,In,Out,Buf,cons,B],
           [N,A,PA,In,Out1,Buf,rem(Buf),B]) :- In>Out, Out1 is Out+1.
trans(cons,[N,A,P,In,Out,Buf,rem(Buf),[Buf|B]],[N,A,P,In,Out,Buf,cons,B]).
trans(err ,[N,A,PA,In,Out,Buf,rem(Buf),[B2|B]],
           [N,err,err,err,err,err,err,err]) :- Buf\==B2.
```

This example cannot be successfully verified by our current approach, due to its inability to detect simple inconsistencies. For example, neither ECCE nor the technique of [30] will detect that the conjunction `X < 1, X > 1` cannot succeed. It should be possible to remedy this deficiency by adding CLP-techniques to ECCE and/or going to more sophisticated abstract domains as outlined in [24]. Similar extensions will probably be needed to handle real time systems [3, 17].

Other problems do appear when we move to formalisms where the state representation gets more complex. For example, whereas for Petri nets a state was just a sequence of natural numbers, in CCS a state is an (arbitrarily complex) expression. As the following example shows, this leads to other difficulties.

Example 2. Take the following simple CCS specification of an agent P (where "a" and "$\bar{a}$" are complementary actions, "." denotes the action prefix, and "|" the parallel composition):

$P =_{Def} a.P|\bar{a}.P$

The transitional semantics of CCS tells us that the agent P can perform the action a and $\bar{a}$ respectively thanks to its left and right branch. Moreover, P can perform the invisible action τ (via the synchronization of a and $\bar{a}$) leading to a new expression $P|P$ which in turn can perform τ leading to $(P|P)|P$ (or $P|(P|P)$). One possible way to encode this system for use by our CTL interpreter is as follows:

```
trans(A,prefix(A,X),X).
trans(A,par(X,Y),par(X1,Y)) :- trans(A,X,X1).
trans(A,par(X,Y),par(X,Y1)) :- trans(A,Y,Y1).
trans(tau,par(X,Y),par(X1,Y1)) :- trans(A,X,X1),trans(bar(A),Y,Y1).
trans(A,agent(X),X1) :- agent(X,XDef),trans(A,XDef,X1).
agent(p, par( prefix(a,agent(p)), prefix(bar(a),agent(p))) ).
```

Having encoded the system, we may wonder whether we can use our approach to prove a very simple safety property: that in no reachable state we can perform an action b (this is indeed an infinite model checking task: the model checker FDR for CSP loops when given such a task). Unfortunately, the present system is incapable of doing so. The problem now is that, as explained above, the state `agent(p)` can lead to the state `par(agent(p),agent(p))` (more precisely the call `sat__0(agent(p))` can lead to `sat__0(par(agent(p),agent(p)))`). This means that when unfolding the interpreter, ECCE which will detect (and rightly so) a possible infinite sequence as `agent(p)` $\trianglelefteq$ `par(agent(p),agent(p))`. The only problem then is that it will compute the most specific generalisation of { `agent(p)` , `par(agent(p),agent(p))` } which is a fresh variable `X`. In other words our approach loses all the information on the system and we cannot prove the safety property (as the unconstrained variable `X` can of course also represent `prefix(b,X)` which *can* perform the action b). So, while the $\trianglelefteq$ relation is quite refined, the most specific generalisation is rather crude and does not take the actual growth information into account. One solution is to generate (regular) types based upon homeomorphic embedding, as outlined in [24]. For this example we would need to generate something like the following type σ for the generalisation: $\sigma =$ `agent(P)` $|$ `par(`σ`,`σ`)` .

6 Going towards full CTL

Let us now examine how we have to adapt our approach to handle more complicated CTL formulae.

Example 3. We can actually prove, using the current tools, the absence of deadlocks ($\forall\Box(\exists\bigcirc true)$) for our Petri net example from Sections 3.1 and 4.4. For this we first apply the LOGEN pre-compilation phase for the formula $\forall\Box(\exists\bigcirc true)$ (i.e., specialising the CTL interpreter for the query `sat(X,ag(en(true))))`):

```
sat__0(B) :- not((B = C, sat__1(C))).
sat__1(B) :- sat_eu__2(B).
sat_eu__2(B) :- not(( B = C,sat__3(C))).
sat_eu__2([s(D),s(E),F,G,H]) :- sat_eu__2([D,E,s(F),G,H]).
sat_eu__2([I,J,s(K),L,M]) :- sat_eu__2([I,s(J),K,s(L),M]).
sat_eu__2([N,O,P,s(Q),R]) :- sat_eu__2([s(N),O,P,Q,s(R)]).
sat__3([s(B),s(C),D,E,F]).
sat__3([G,H,s(I),J,K]).
sat__3([L,M,N,s(O),P]).
```

Observe that this program now contains negations and that both the ECCE system and the technique of [30] only provide a safe but rather crude treatment of negation (we will actually extend [30] slightly below).

Applying the ECCE partial deduction for the initial marking $\langle 1, 1, 0, 0, 0\rangle$ (i.e., specialising `sat__0([s(0),s(0),0,0,0])`) now gives:

```
sat__0([s(0),s(0),0,0,0]) :- not(sat__1__2).
sat__0__1 :- not(sat__1__2).
sat__1__2 :- not(sat__3__3).          sat_eu__2__6 :- not(sat__3__7).
sat__1__2 :- sat_eu__2__4.            sat_eu__2__6 :- sat_eu__2__8.
sat__3__3.                            sat__3__7.
sat_eu__2__4 :- not(sat__3__5).       sat_eu__2__8 :- not(sat__3__9).
sat_eu__2__4 :- sat_eu__2__6.         sat_eu__2__8 :- sat_eu__2__4.
sat__3__5.                            sat__3__9.
```

Let us now compute the most specific version of this residual program. In order for the example to go through we actually have to extend [30] by adding a rudimentary treatment of negation by extracting, as explained in Section 4.4, a safe under-approximation of the success set from the residual program and using it to obtain a safe over-approximation of negated calls: we will (correctly) assume that $not(p(\bar{t})\theta)$ fails if there is a fact $p(\bar{t}) \leftarrow$ in the program. Otherwise, we assume that a negated call potentially succeeds. We then obtain:

```
sat__0([s(0),s(0),0,0,0]) :- not(sat__1__2).
sat__0__1 :- not(sat__1__2).
sat__1__2 :- fail.                   sat_eu__2__6 :- fail.
sat__3__3.                           sat__3__7.
sat_eu__2__4 :- fail.                sat_eu__2__8 :- fail.
sat__3__5.                           sat__3__9.
```

This already contains the information that our system satisfies the CTL formula, but re-applying ECCE once more makes this fully explicit:

```
sat__0([s(0),s(0),0,0,0]).
```

Similarly, we can try to prove the absence of deadlocks for *any* number of processes ≥ 1. For this we specialised for `sat__0([s(A),s(0),0,0,0])` using ECCE 3 times (using a more agressive unfolding rule) interleaved with two (extended) most specific version computations, thereby obtaining:

```
sat__0([s(A),s(0),0,0,0]).
```

If we try to prove that absence of deadlocks holds for any number of processes even 0, by specialising the call `sat__0([A,s(0),0,0,0])`, we obtain after 4 iterations:

```
sat__0([A,s(0),0,0,0]) :- not(sat__1__2__2__2__2(A)).
sat__0__1(A) :- not(sat__1__2__2__2__2(A)).
sat__1__2__2__2__2(A) :- not(sat__3__5__4__3__3(A)).
sat__3__5__4__3__3(s(A)).
```

I.e. `sat__1__2__2__2__2(A)` is true for $A = 0$ and thus `sat__0([A,s(0),0,0,0])` is false for $A = 0$ and we have identified the counter-example. We have also re-proven that for any value > 0 the property holds.

One can notice that, as the systems and properties get more complex, more and more iterations of ECCE and most specific version computations are required. For more complicated examples we will probably reach the limit of an approach working by *separate* phases and we will need the *fully integrated* techniques of [25, 24] (there are certain properties which can only be proven by a fully integrated approach, see [25]). Also, for more complicated CTL-formulae, the above treatment of negation will be too rudimentary and we will need more refined under-approximations of the success set. Something along the lines of constructive negation, allowing to extract partial answers from a negated call, might also prove to be essential. All of this is subject of ongoing research.

7 Conclusion, Assessment, and Future Work

We have shown the usefulness of logic programming techniques for model checking. We have presented a complete interpreter for CTL formulae, implemented as a pure logic program (e.g., without the `tfindall` used in [35, 28]) and we have shown it to be correct (under the SLS/well-founded semantics), even for infinite state systems. We have shown how this interpreter can be used for finite state model checking using tabling-based execution. We also have presented a particular technique for infinite state model checking of safety properties, using existing techniques for partial deduction and abstract interpretation, as implemented in the ECCE system. The idea was to reduce the interpreter, searching for unsafe states, to the empty program. We discussed how this approach has to be extended to handle more complicated infinite state systems and to handle arbitrary CTL formulae. We presented some succesfull examples but argue that more refined treatment of negation and more refined abstract domains will be required for the method to scale up to such systems and properties.

Of course, an important aspect of model checking of finite state systems is the complexity of the underlying algorithms. We have not touched upon this issue in the present paper, but plan to do so in future work. In future work, we will also strive to identify classes of problems and infinite state systems which can be precisely solved by our approach. First promising results, for coverability problems of unbounded Petri nets, have been obtained.

Another important issue arises when our model checking approach is incapable of establishing the desired property. In that case, one would like to assist the user by extracting a counter example from the residual program (if possible; due to the undecidability of most problems for infinite state systems we actually cannot be sure whether such a counter example exists). A naive solution is to run the residual program using some sophisticated computation mechanisms such as tabling, breadth-first, or iterative deepening.

References

1. K. Apt and E. Olderog. *Verification of Sequential and Concurrent Programs.* Springer-Verlag, 1991.

2. S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems compositionally and automatically. In *Proceedings of CAV'98*, LNCS, pages 319–331. Springer-Verlag, 1998.
3. 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. Extended version as Research Report LSV-99-3, Lab. Specification and Verification, ENS de Cachan, Cachan, France, Mar. 1999.
4. A. Bossi, M. Gabrielli, G. Levi, and M. Martelli. The s-semantics approach: Theory and applications. *The Journal of Logic Programming*, 19 & 20:149–198, May 1994.
5. R. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.
6. O. Burkart and J. Ezparza. More infinite results. In *Proceedings of Infinity'96*, 1996. Research Report MIP-9614, University of Passau.
7. W. Charatonik and A. 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.
8. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.
9. E. M. Clarke and J. M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, Dec. 1996.
10. D. De Schreye, R. Glück, J. Jørgensen, M. Leuschel, B. Martens, and M. H. Sørensen. Conjunctive partial deduction: Foundations, control, algorithms and experiments. *The Journal of Logic Programming*, 41(2 & 3):231–277, 1999.
11. 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*, LNCS 131, pages 52–71, Yorktown Heights, New York, May 1981. Springer-Verlag.
12. J. Ezparza. Decidability of model-checking for infinite-state concurrent systems. *Acta Informatica*, 34:85–107, 1997.
13. J. Gallagher and M. Bruynooghe. The derivation of an algorithm for program specialisation. *New Generation Computing*, 9(3 & 4):305–333, 1991.
14. R. Glück and M. Leuschel. Abstraction-based partial deduction for solving inverse problems – a transformational approach to software verification. In *Proceedings of PSI'99*, LNCS 1755, pages 93–100, Novosibirsk, Russia, 1999. Springer-Verlag.
15. P. Hartel, M. Butler, A. Currie, P. Henderson, M. Leuschel, A. Martin, A. Smith, U. Ultes-Nitsche, and B. Walters. Questions and Answers About Ten Formal Methods. Proceedings of FMICS'99, Trento, Italy, 1999.
16. J. Hatcliff, M. Dwyer, and S. Laubach. Staging analysis using abstraction-based program specialization. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Proceedings of ALP/PLILP'98*, LNCS 1490, pages 134–151. Springer-Verlag, 1998.
17. T. A. Henzinger and P.-H. Ho. HYTECH: The Cornell HYbrid TECHnology tool. *LNCS*, 999:265–293, 1995.
18. G. Higman. Ordering by divisibility in abstract algebras. *Proceedings of the London Mathematical Society*, 2:326–336, 1952.
19. J. Jørgensen and M. Leuschel. Efficiently generating efficient generating extensions in Prolog. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proceedings of the 1996 Dagstuhl Seminar on Partial Evaluation*, LNCS 1110, pages 238–262, Schloß Dagstuhl, 1996. Springer-Verlag.
20. J. B. Kruskal. Well-quasi ordering, the tree theorem, and Vazsonyi's conjecture. *Transactions of the American Mathematical Society*, 95:210–225, 1960.

21. J.-L. Lassez, M. Maher, and K. Marriott. Unification revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan-Kaufmann, 1988.
22. M. Leuschel. Improving homeomorphic embedding for online termination. In P. Flener, editor, *Proceedings of LOPSTR'98*, LNCS 1559, pages 199–218, Manchester, UK, June 1998. Springer-Verlag.
23. M. Leuschel. On the power of homeomorphic embedding for online termination. In G. Levi, editor, Static Analysis. *Proceedings of SAS'98*, LNCS 1503, pages 230–245, Pisa, Italy, September 1998. Springer-Verlag.
24. M. Leuschel. Program specialisation and abstract interpretation reconciled. In J. Jaffar, editor, *Proceedings of JICSLP'98*, pages 220–234, Manchester, UK, June 1998. MIT Press.
25. M. Leuschel and D. De Schreye. Logic program specialisation: How to be more specific. In H. Kuchen and S. Swierstra, editors, *Proceedings of PLILP'96*, LNCS 1140, pages 137–151, Aachen, Germany, September 1996. Springer-Verlag.
26. 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.
27. 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.
28. X. Liu, C. R. Ramakrishnan, and S. A. Smolka. Fully local and efficient evaluation of alternating fixed points. In B. Steffen, editor, *Proceedings of TACAS'98*, LNCS 1384, pages 5–19. Springer-Verlag, 1998.
29. J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11(3& 4):217–242, 1991.
30. K. Marriott, L. Naish, and J.-L. Lassez. Most specific logic programs. *Annals of Mathematics and Artificial Intelligence*, 1:303–338, 1990.
31. K. L. McMillan. *Symbolic Model Checking*. PhD thesis, Boston, 1993.
32. F. Moller. Infinite results. In *Proceedings of CONCUR'96*, LNCS 1119, pages 195–216. Springer-Verlag, 1996.
33. U. Nitsche and P. Wolper. Relative liveness and behavior abstraction. In *Proceedings of PODC'97*, pages 45–52, Santa Barbara, California, 1997. ACM.
34. T. C. Przymusinksi. On the declarative and procedural semantics of logic programs. *Journal of Automated Reasoning*, 5(2):167–205, 1989.
35. Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. Swift, and D. S. Warrend. Efficient model checking using tabled resolution. In *Proceedings of CAV'97*, LNCS. Springer-Verlag, 1997.
36. J. Rushby. Mechanized formal methods: Where next? In *Proceedings of FM'99*, LNCS 1708, pages 48–51, Toulouse, France, Sept. 1999. Springer-Verlag.
37. 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.
38. M. H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In J. W. Lloyd, editor, *Proceedings of ILPS'95*, pages 465–479, Portland, USA, December 1995. MIT Press.
39. P. Wolper and B. Boigelot. Verifying systems with infinite but regular state spaces. *Proceedings of CAV'98*, LNCS 1427, pages 88–97. Springer-Verlag, 1998.