Solving Coverability Problems of Petri Nets by Partial Deduction*

Michael Leuschel Helko Lehmann[†]
Department of Electronics and Computer Science
University of Southampton
Southampton SO17 1BJ, U.K.
{mal,hel99r}@ecs.soton.ac.uk

ABSTRACT

In recent work it has been shown that infinite state model checking can be performed by a combination of partial deduction of logic programs and abstract interpretation. This paper focuses on a particular class of problems—coverability for (infinite state) Petri nets—and shows how existing techniques and tools for declarative programs can be successfully applied. In particular, we show that a restricted form of partial deduction is already powerful enough to decide all coverability properties of Petri Nets. We also prove that two particular instances of partial deduction exactly compute the Karp-Miller tree as well as Finkel's minimal coverability set. We thus establish a link between algorithms for Petri nets and logic program specialisation.

Keywords

Program analysis and verification; logic programming; model checking; concurrency; partial evaluation; abstract interpretation; Petri nets.

1. INTRODUCTION AND SUMMARY

Recently there has been interest in applying logic programming techniques to model checking. Table-based logic programming, set-based analysis, and constraint logic programming can be used as an efficient means of performing model checking [33] [4] [6, 13]. Despite the success of model checking, most systems must still be substantially simplified and considerable human ingenuity is required to arrive at the stage where the push button automation can be applied [35]. Furthermore, most *software* systems cannot be modelled directly by a *finite* state system: as soon as some kind of recursion or sophisticated data structures come into play, an infinite number of states must be verified. For these reasons, there has recently been considerable interest in *infinite model checking* (e.g., [1, 31, 10, 9, 39, 6]). This, by its very undecidable nature, is a daunting task, for which *abstraction* is a key issue. Indeed, abstraction allows one to approximate an infinite system by a finite one, and if proper care is taken the results obtained for the finite abstraction will be valid for the infinite system.

An important question when attempting to perform infinite model checking in practice is: How can one *automatically* obtain an abstraction which is finite, but still as precise as required? A potential solution to this problem is to apply existing techniques for the *automatic* control of *(logic) program specialisation* [23]. More precisely, in program specialisation in general and partial deduction[1] in particular, one faces a very similar (and quite extensively studied) problem: To be able to produce efficient specialised programs, *infinite* computation trees have to be abstracted in a *finite* but also as *precise* as possible way.

The simplest way to apply this existing technology is to model the system to be verified as a logic program (by means of an interpreter). This translation is often very straightforward (e.g., due to the built-in support for non-determinism) and enables us to express infinite state systems. First successful steps in that direction have been taken in [14, 26], but up to now there were no results for what systems and properties this approach yields decision procedures and how it relates to existing model checking algorithms. In this paper we give the first formal answer to these questions, and show that when we encode Petri nets as logic programs and use "typical" existing program specialisation algorithms, we get decision procedures for the so-called "coverability problems" (which encompass quasi-liveness, boundedness, determinism, regularity,...). Moreover, quite surprisingly, we can exactly mimic a well known Petri net algorithm by Karp and Miller when slightly weakening the program specialisation techniques. These insights do not only shed light on the power of using logic program analysis and specialisation for infinite state model checking; they also establish a link between algorithms in Petri net theory and program specialisation and will hopefully lead to further insights and cross-

*(Produces the permission block, copyright information and page numbering). For use with ACM_PROC_ARTICLE-SP.CLS V2.0. Supported by ACM.

[†]The author acknowledges support from the EPSRC under grant no. 99308892.

[1]Partial evaluation of logic programs is often referred to as partial deduction.

fertilization. Already in this paper an extension of partial deduction was inspired by another algorithm by Finkel [11].

The paper is structured as follows. In Section 2 we introduce Petri nets, coverability problems and the Karp-Miller procedure. In Section 3 we present partial deduction of logic programs, along with a generic and two concrete algorithms. In Section 4 we show how Petri nets can be encoded as logic programs which enables us in Section 5 to apply partial deduction algorithms to Petri nets and establish the relationship to coverability problems and algorithms. Finally, in Section 6 we discuss how the partial deduction approach can deal with more complicated systems and properties.

2. PETRI NETS AND COVERABILITY

In this paper we want to study the power of partial deduction based approaches for model checking of infinite state systems. To arrive at precise results, it makes sense to focus on well-established classes of infinite state systems and properties which are known to be decidable. One can then examine whether the partial deduction approach provides a decision procedure and how it compares to existing algorithms. In this section, we describe such a decidable class of systems and properties, namely Petri nets [34] and coverability problems. We start out by giving definitions of some important concepts in Petri net theory.

Definition 1. A *Petri net* Π is a tuple (S, T, F, M_0) consisting of a finite set of places S, a finite set of transitions T with $S \cap T = \emptyset$ and a flow relation F which is a function from $(S \times T) \cup (T \times S)$ to $\mathbb{N}$. A *marking* m for Π is a mapping $S \mapsto \mathbb{N}$. M_0 is a marking called *initial*.

A transition $t \in T$ is *enabled* in a marking M iff $\forall s \in S : M(s) \geq F(s,t)$. An enabled transition can be fired, resulting in a new marking M' defined by $\forall s : M'(s) = M(s) - F(s,t) + F(t,s)$. We will denote this by $M[t\rangle M'$. By $M[t_1, \ldots, t_k\rangle M'$ we denote the fact that for some intermediate markings $M_1, \ldots, M_{k-1}$ we have $M[t_1\rangle M_1, \ldots, M_{k-1}[t_k\rangle M'$.

We define the reachability tree $RT(\Pi)$ inductively as follows: Let M_0 be the label of the root node. For every node n of $RT(\Pi)$ labelled by some marking M and for every transition t which is enabled in M, add a node n' labelled M' such that $M[t\rangle M'$ and add an arc from n to n' labelled t. The set of all labels of $RT(\Pi)$ is called the *reachability set* of Π, denoted $RS(\Pi)$. The set of words given by the labels of finite paths of $RT(\Pi)$ starting in the root node is called *language* of Π, written $L(\Pi)$.

For convenience, we denote $M \geq M'$ iff $M(s) \geq M'(s)$ for all places $s \in S$. We also introduce *pseudo-markings*, which are functions from S to $\mathbb{N} \cup \{\omega\}$ where we also define $\forall n \in \mathbb{N} : \omega > n$ and $\omega + n = \omega - n = \omega + \omega = \omega$. Using this we also extend the notation $M_{k-1}[t_1, \ldots, t_k\rangle M'$ for such markings.

Finally, a sequence of markings $M_1, M_2, \ldots$ is said to *converge* to the pseudo-marking M' when, for every place $s \in S$ and for every integer $n \geq 1$, there exists k such that $\forall k' > k$ we have $M_{k'}(s) \geq n$ if $M'(s) = \omega$ otherwise we have $M_{k'}(s) = M'(s)$.[2]

Many interesting properties of Petri nets can be investigated using the so-called *Karp-Miller tree* resulting from the algorithm[3] below, first defined in [19]. The Karp-Miller tree is a finite abstraction of the set of reachable markings $RS(\Pi)$ with which we can decide whether it is possible to "cover" some arbitrary marking M' (in the sense that $\exists M'' \in RT(\Pi) \mid M'' \geq M'$) simply by checking whether a node M''' in the tree covers M' (i.e., $M''' \geq M'$). The main idea of the algorithm is to simulate the execution of a Petri net Π, starting from the initial marking M_0, until one reaches a marking which is greater or equal than a preceding one. If this marking is identical to a predecessor, i.e., it has already been dealt with before, then the algorithm will not treat it again. If this marking is strictly greater than a predecessor, however, then one will generalise the marking by putting ω's into all places where the number of tokens has actually increased. Simulation will then proceed using this new pseudo-marking. For example, if one can reach a marking $M_2 = \langle 1,3,2,2 \rangle$ from $M_1 = \langle 1,0,2,1 \rangle$ by firing a certain sequence of transitions $t_1, \ldots, t_n$, the algorithm will continue the simulation from the pseudo-marking $\langle 1, \omega, 2, \omega \rangle$ and not from M_2. The justification is that, due to monotonicity of Petri nets, we can repeatedly fire the sequence $t_1, \ldots, t_n$ to generate arbitrarily large number of tokens in the places 2 and 4.

In Algorithm 2.1 we describe precisely how to compute the Karp-Miller tree $KM(\Pi)$. Nodes in the tree are couples (k, M), where k is a unique identifier and M is a pseudo-marking. We also say that (k, M) is labelled by M.

Algorithm 2.1 (*Karp–Miller–Tree*)

Input: a Petri net $\Pi = (S, T, F, M_0)$
Output: a tree $KM(\Pi)$ of nodes labelled by pseudo-markings
Initialisation: set $U := \{(r, M_0)\}$ of unprocessed nodes
while $U \neq \emptyset$
  select some $(k, M) \in U$;
  $U := U \setminus \{(k, M)\}$;
  if there is no ancestor node (k_1, M_1) of (k, M) with $M = M_1$ **then**
    $M_2 = M$;
    for all ancestors (k_1, M_1) of (k, M) such that $M_1 < M$ **do**
      for all places $p \in S$ such that $M_1(p) < M(p)$ **do** $M_2(p) = \omega$;
    $M := M_2$;
    for every transition t such that $M[t\rangle M'$ **do**
      create node (k', M');
      create arc labelled t from (k, M) to (k', M');
      $U := U \cup (k', M')$;

[2] This deviates slightly from the definition in [11], according to which the sequence $\langle 0,0 \rangle, \langle 1,0 \rangle, \langle 1,1 \rangle, \langle 2,1 \rangle, \ldots$ does not converge to $\langle \omega, \omega \rangle$.

[3] The algorithm presented here differs slightly from the original.

In the following we list properties of Petri nets which are known to be decidable using the Karp-Miller-tree $KM(\Pi)$, and we sketch the respective decision procedures (see [11],[19], [38]). The Karp-Miller-tree is always finite, hence it is possible to check for properties of all arcs and all nodes of $KM(\Pi)$.

1. Is $RT(\Pi)$ of a Petri net Π infinite? **Procedure:** $RT(\Pi)$ is infinite iff the label of some node of $KM(\Pi)$ contains an ω, or, some node has an ancestor labelled by the same marking.
2. Is $RS(\Pi)$ of a Petri net Π infinite? **Procedure:** $RS(\Pi)$ is infinite iff the label of some node of $KM(\Pi)$ contains an ω.
3. Is a place $s \in S$ of a Petri net $\Pi = (S, T, F, M_0)$ bounded? **Procedure:** s is bounded iff no node of $KM(\Pi)$ is labelled by a pseudo-marking M such that $M(s) = \omega$.
4. Is a transition $t \in T$ of a Petri net $\Pi = (S, T, F, M_0)$ quasi–live, i.e. is there a marking in $RT(\Pi)$ where t is enabled? **Procedure:** t is quasi–live iff there is a pseudo-marking M in $KM(\Pi)$ such that $M(s) \geq F(s,t)$ for all $s \in S$.
5. Does $RS(\Pi)$ of a Petri net $\Pi = (S, T, F, M_0)$ cover some marking M'? **Procedure:** M' is covered iff there is a pseudo-marking M in $KM(\Pi)$ such that $M \geq M'$.
6. Is $L(\Pi)$ of a Petri net Π regular? **Procedure:** $L(\Pi)$ is regular iff for every node n of $KM(\Pi)$ which is labelled by a marking containing ω and for every ancestor n_a of n such that n_a is labelled by the same marking, we have that $\sum_{t \in w}(|w|_t(F(t,s) - F(s,t))) \geq 0$ for all $s \in S$, where $w \in T^+$ is the path in $KM(\Pi)$ from n_a to n and $|.|_t$ denotes the Parikh–mapping (i.e., $|w|_t$ is the number of t's in w).

Several other problems (e.g., control state reachability, determinism) can be reduced to one of the above [11]. However, the Karp-Miller-tree of a Petri net can become very large (in general the size of the graph is not bounded by a primitive recursive function [32]). But, it was shown by Finkel [11] that the above problems can also be decided using minimal coverability sets and graphs, which are often significantly smaller.

Definition 2. A *coverability set* $CS(\Pi)$ of a Petri net $\Pi = (S, T, F, M_0)$ is a set of pseudo-markings such that the following conditions hold:

1. for every reachable marking $M \in RS(\Pi)$, there is a marking $M' \in CS(\Pi)$ such that $M \leq M'$,
2. for every marking $M' \in CS(\Pi) \setminus RS(\Pi)$, there is an infinite strictly increasing sequence of reachable markings $\{M_n\}$ converging to M'.

A coverability set $CS(\Pi)$ is minimal iff no proper subset of $CS(\Pi)$ is a coverability set of Π.

The minimal coverability set is finite and unique [11]. In fact, the finite reachability set problem, the boundedness problem, the covering problem, and the quasi–liveness problem are decidable using coverability sets, only. To decide the regularity and the finite reachability tree problem, coverability graphs can be used [11].

3. PARTIAL DEDUCTION

We will now present the essential ingredients of the logic program specialisation techniques that were used for infinite model checking in [14, 26]. As it turns out, we do not require a separate abstract interpretation phase (as in [14, 26]) to achieve completeness; partial deduction alone is already powerful enough. We will return to the issue of implementing Petri nets as logic programs (and thus making them amenable to partial deduction) in Section 4.

Throughout this article, we denote variables through (strings starting with) an uppercase symbol, while constants, functions, and predicates begin with a lowercase character.

A *partial evaluator* [17] is given a program P along with *part* of its input, called the *static input*. The partial evaluator then produces a correct *specialised* version P_S of P which, when given the dynamic input, produces the same output as the original program P. In logic programming full input to a program P consists of a goal $\leftarrow Q$ and evaluation corresponds to constructing a complete SLD-tree for $P \cup \{\leftarrow Q\}$, i.e., a tree whose root is labelled by $\leftarrow Q$ and where children of nodes are obtained by first selecting a literal of the node and then resolving it with the clauses of P. For partial evaluation, static input takes the form of a *partially instantiated* goal $\leftarrow Q'$ and the specialised program P_S should be correct for all runtime instances $\leftarrow Q'\theta$ of $\leftarrow Q'$ in the sense that the computed answers of $P \cup \{\leftarrow Q'\theta\}$ and $P_S \cup \{\leftarrow Q'\theta\}$ are identical. A technique which achieves this is known under the name of *partial deduction*, which we present below.

3.1 Generic Algorithm for Partial Deduction

The general idea of partial deduction is to construct a finite number of finite but possibly incomplete[4] trees which "cover" the possibly infinite SLD-tree for $P \cup \{\leftarrow Q'\}$ (and thus also all SLD-trees for all instances of $\leftarrow Q'$). The derivation steps in these SLD-trees are the computations which have been pre-evaluated and the clauses of the specialised program are then extracted by constructing one specialised clause (called a *resultant*) per branch. These incomplete SLD-trees are obtained by applying an unfolding rule:

Definition 3. An *unfolding rule* is a function which, given a program P and a goal $\leftarrow Q$, returns a non-trivial[5] and possibly incomplete SLD-tree τ for $P \cup \{\leftarrow Q\}$. We also define the set of leaves, *leaves*(τ), to be the leaf goals of τ.

Formally, the resultant of a branch of τ leading from the root $\leftarrow Q$ to a leaf goal $\leftarrow Q_i$ via computed answer θ_i is the

[4] An *incomplete* SLD-tree is a SLD-tree which, in addition to success and failure leaves, also contains leaves where no literal has been selected for a further derivation step.

[5] A trivial SLD-tree has a single node where no literal has been selected for resolution.

formula $Q\theta_i \leftarrow Q_i$. 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). Given *closedness* (all leaves are an instance of a specialised atom) and *independence* (no two specialised atoms have a common instance), correctness of the specialised program is guaranteed [28]. Independence is usually (e.g. [25, 5]) ensured by a *renaming* transformation which maps dependent atoms to new predicate symbols (and often filters out constant parts as well). Closedness is more difficult to ensure, but can be satisfied using the following generic algorithm based upon [25]. This algorithm structures the atoms to be specialised in a *global tree*: i.e., a tree whose nodes are labelled by atoms and where A is a descendant of B iff specialising B lead to the specialisation of A. Apart from the missing treatment of conjunctions [5] the following is the algorithm implemented in the ECCE system [21] which we will employ later on.

Algorithm 3.1 (*generic partial deduction algorithm*)

Input: a program P and a goal $\leftarrow A$

Output: a set of atoms or conjunctions $\mathcal{A}$ and a global tree γ

Initialisation: γ := a "global" tree with a single unmarked node, labelled by A

repeat
  pick an unmarked or abstracted leaf node L in γ
  if $covered(L,\gamma)$ **then** mark L as processed
  else
    $W = whistle(L,\gamma)$
    if $W \neq fail$ **then**
      if $parent\_abstraction = true$ **then**
        remove all descendants of W and mark W as abstracted
        $label(W) := abstract(L, W, \gamma)$
      else
        mark L as abstracted
        $label(L) := abstract(L, W, \gamma)$
    else
      mark L as processed
      for all $Q \in leaves(U(P, label(L)))$ **do**
        for all $A \in partition(Q)$ **do**
          add a new unmarked child C of L to γ
          $label(C) := A$
until all nodes are processed
output $\mathcal{A} := \{label(A) \mid A \in \gamma\}$ and γ

As can be seen, the algorithm is parametrised by a boolean constant *parent_abstraction*, an unfolding rule U, a partitioning function *partition*, a predicate $covered(L,\gamma)$, a whistle function $whistle(L,\gamma)$ and an abstraction function $abstract(L,W,\gamma)$.

Intuitively, $covered(L,\gamma)$ is a way of checking whether L or a generalisation of L has already been treated in the global tree γ. Formally, $covered(L,\gamma) = true$ must imply that $\exists M \in \gamma$ such that M is processed or abstracted and for some substitution θ: $label(M)\theta = label(L)$. A particular implementation can decide to be more demanding and, e.g., return true only if there is another node in γ labelled by a variant of L. The purpose of *partition* is to divide leaf goals into individual atoms. For "classical" partial deduction we would simply define: $partition(\leftarrow A_1,...,A_n) = \{A_1,...,A_n\}$.[6] However, it is possible to perform some abstraction—independent of the particular global tree γ— at that stage. Formally, whenever $partition(\leftarrow Q) = \{A_1,...,A_k\}$ then for some $\theta_1,...,\theta_k$ we must have that $\leftarrow Q$ is identical to $\leftarrow A_1\theta_1,...,A_n\theta_n$ (up to reordering of atoms).

The other two parameters are used to ensure termination. Intuitively, the $whistle(L,\gamma)$ is used to detect whether the branch of γ ending in L is "dangerous", in which case it returns a value different from *fail* (i.e., it "blows"). This value should be an ancestor W of L compared to which L looked dangerous (e.g., L is bigger than W in some sense). The abstraction operation will then compute a generalisation of L and W, less likely to lead to non-termination. Formally, $abstract(L,W,\gamma)$ must be an atom which is more general than both L and W. Depending on the parameter *parent_abstraction* this generalisation will either replace the label of W or L in the global tree γ.

If the Algorithm 3.1 terminates then the closedness condition of [28] is satisfied, i.e., it is ensured that *together* the SLD-trees $\tau_1,\ldots,\tau_n$ form a *complete description* of all possible computations that can occur for all concrete instances $\leftarrow A\theta$ of the goal of interest [22]. We can then produce a *totally correct* specialised program.

Note that Algorithm 3.1 can be seen as a special kind of *forwards* abstract interpretation (see [22]), 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).

On its own, Algorithm 3.1 does not ensure termination (so its strictly speaking not an algorithm but a procedure). To ensure termination, we have to use an unfolding rule that builds finite SLD-trees only. We also have to guarantee that infinite branches in the global tree γ will be spotted by the *whistle* and that the abstraction can not be repeated infinitely often.

3.2 Concrete Algorithms

We now present two concrete partial deduction algorithms. These algorithms are *online* (as opposed to *offline*) in the sense that they take their control decisions *during* the construction of γ and not beforehand. They are also rather naïve (e.g., they do not use characteristic trees [25] nor do they cater for conjunctive partial deduction [5]; also the generic Algorithm 3.1 does not include recent improvements such as constraints or abstract interpretation). However, they are easier to comprehend (and analyse) and will actually be sufficiently powerful to solve several interesting problems.

Unfolding Rule

In this paper we will use a very simple method for ensuring that each individual SLD-tree constructed by U is finite: we always do just a single unfolding step! Practical systems use much more refined approaches.

[6] For conjunctive partial deduction one can return entire conjunctions instead of just single atoms.

Whistle

To ensure that no infinite global tree γ is being built-up, we will use a more refined approach based upon well-quasi orders:

Definition 4. A sequence $s_1, s_2, \ldots$ of elements of S is called *admissible wrt a binary relation* $\leq_S$ on $S \times S$ iff there are no $i < j$ such that $s_i \leq_S s_j$. We say that $\leq_S$ is a *well-quasi relation* (wqr) iff there are no infinite admissible sequences wrt $\leq_S$. A *well quasi order* (wqo) is a reflexive and transitive wqr.

In our context we will use a wqo to ensure that no infinite tree γ is built up in Algorithm 3.1 by setting $whistle \neq fail$ whenever the sequence of labels on the current branch is not admissible.

Definition 5. Given a wqo $\preceq$ we define $whistle_{\preceq}$ as follows: $whistle_{\preceq}(L, \gamma) = M$ iff M is the farthest ancestor of L such that $label(M) \preceq label(L)$ and $whistle_{\preceq}(L, \gamma) = fail$ if there is no such ancestor.

A particularly useful wqo is the homeomorphic embedding [16, 20]. The following is the definition from [37], which adapts the pure homeomorphic embedding from [7] by adding a rudimentary treatment of variables.

Definition 6. 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 $\forall i \in \{1, \ldots, n\}$ we have $s_i \trianglelefteq t_i$.

Notice that n is allowed to be 0 and we thus have $c \trianglelefteq c$ for all constant and proposition symbols. The intuition 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 $f(a, b) \trianglelefteq p(f(g(a), b))$ but $p(a) \not\trianglelefteq p(b)$ and $f(a, b) \not\trianglelefteq p(f(a, c), f(c, b))$.

The relation $\trianglelefteq$ is a wqo on the set of expressions over a finite alphabet [16, 20]. We also define the *weak homeomorphic embedding* relation $\trianglelefteq^-$ on expressions by replacing rule 1. of Definition 6 by:

1. $t \trianglelefteq X$ for all variables X

$\trianglelefteq^-$ is weaker in the sense that less sequences are admissible (and hence it is a wqr; it is not a wqo as it is not transitive), but it will be useful in establishing exact relationships to existing algorithms for Petri nets.

Abstraction

Once the *whistle* has identified a potential non-termination one will usually compute generalisations which are as precise as possible (for partial deduction): The *most specific generalisation* of a finite set of expressions S, denoted by $msg(S)$, is the most specific expression M such that all expressions in S are instances of M. A much less precise generalisation than the msg, which we henceforth call $nmsg$ (naïve msg), is defined as follows on terms:

- $nmsg(s, t) = s$ if $s = t$ and both s and t are ground terms
- $nmsg(s, t) = X$ otherwise, where X is a fresh variable

For two atoms $p(s_1, \ldots, s_k)$ and $p(t_1, \ldots, t_k)$ we define:
- $nmsg(p(s_1, \ldots, s_k), p(t_1, \ldots, t_k)) =$
$p(nmsg(s_1, t_1), \ldots, nmsg(s_k, t_k))$.

This will always produce a more general expression, but not the most specific one. Note that $nmsg$ is both associative and commutative, and we can therefore extend it to sets of atoms by defining:
- $nmsg(\{A_1, ..., A_k\}) = nmsg(\ldots nmsg(A_1, A_2) \ldots), A_k)$

We can also use $nmsg$ to implement a naïve partitioning function (where $A' \in Q$ means that A' is an atom in Q):

$naive\_partition(Q) = \{A \mid A' \in Q \text{ and } A = nmsg(A', A')\}$.

This partitioning function will split any conjunction into individual atoms and will replace non-ground terms by a fresh variable. For example, we have that $naive\_partition(\leftarrow p(0), q(s(X), X, s(0))) = \{p(0), q(V, W, s(0))\}$.

Due to their naïvety, neither $nmsg$ nor $naive\_partition$ are used (to our knowledge) in existing online[7] partial deduction systems, but it will allow us to more easily establish precise relationships to algorithms over Petri nets.

Putting it all together

Based upon the concepts we can now define our first concrete partial deduction algorithm:

Algorithm 3.2 We define an instance of Algorithm 3.1 obtained by using:

- $parent\_abstraction = false$
- U unfolds just once
- $partition(Q) = naive\_partition(Q)$
- $covered(L, \gamma) = true$ if there exists an *ancestor* node of L in γ which is different from L and whose label is a *variant* of $label(L)$
- $whistle(L, \gamma) = fail$ if L is marked as abstracted and $whistle(L, \gamma) = whistle_{\trianglelefteq^-}(L, \gamma)$ otherwise
- $abstract(L, W, \gamma) = nmsg(\{label(M_1), \ldots, label(M_k)\})$ where $\{M_1, \ldots, M_k\}$ are all the ancestors of L with $label(M_i) \trianglelefteq^- label(L)$.

From a partial deduction perspective this algorithm is sub-optimal: *covered* only looks at ancestors (normally one would look anywhere in the tree), *partition* and *nmsg* explicitly throw information away, the *whistle* is much cruder than in most partial deduction algorithms (and is also "disabled" on already abstracted nodes, which in general could endanger termination [but it does not do so here]). Its interest is that it mimics exactly the construction of the so-called

[7] *naive_partition* is quite close to what offline systems (such as [18]) do.

Karp-Miller coverability tree, which will enable us to prove completeness results for partial deduction in Section 5.

The following algorithm is a more "natural" partial deduction variant,in which *parent_abstraction*, *covered* and *whistle* are modified.

Algorithm 3.3 We define an instance of Algorithm 3.1 by modifying the concrete Algorithm 3.2 as follows:

- $parent\_abstraction = true$
- $covered(L, \gamma) = true$ if there exists a processed node in γ whose label is *more general than* $label(L)$
- $whistle(L, \gamma) = whistle_{\trianglelefteq^-}(L, \gamma)$
- in addition, after Algorithm 3.1 is complete: remove all nodes L (and their subtrees) for which $covered(L, \gamma) = true$.

This algorithm is more natural in the sense that *covered* looks for variants (and more general atoms) in the *entire* tree; the whistle is also more natural as it is not disabled for already abstracted nodes. Also, when the *whistle* blows the parent node gets abstracted which usually results in much shorter, and more natural specialised programs.

PROPOSITION 1. *Algorithm 3.3 terminates for any program P and goal $\leftarrow A$.*

PROOF. This algorithm is a much simplified version of the algorithms in [25] and termination can be proven in a straightforward manner by simplified version of the proofs in [25] or using the termination framework of [36]. □

For Algorithm 3.2 the *whistle* is disabled for already abstracted atoms. This actually means that the algorithm does not always terminate! However, it will terminate for Petri nets encoded as logic programs.[8] We will return to this issue later in the paper.

4. PETRI NETS AS LOGIC PROGRAMS

It is very easy to implement Petri nets as (non–deterministic) logic programs. Figure 2 contains a particular encoding of the Petri Net on the right in Figure 1 (taken from [11]) and a simple predicate `reachable` searching for reachable markings in $RS(\Pi)$. Other Petri Nets can be encoded by changing the `trans/3` facts and the `initial_marking` fact. Based upon such a translation, [14, 26] pursued the idea that model checking of safety properties amounts to showing that there exists no trace which leads to an unsafe state. In other words, one can translate the interpreter in Fig. 2 into a simple model checker of safety properties by adding a condition to the fact `reachable([],State,State)` which detects unsafe states. E.g., if we want to verify that there should never be tokens in the places $p2$ and $p4$ at the same time, we would write in Fig. 2:

```
reachable([],State,State) :-
                State = [P1,s(P2),P3,s(P4),P5].
```

Proving that no trace leads to a state where `reachable/3` holds is then achieved by a *semantics-preserving program specialisation and analysis technique*, reducing the predicate `reachable/1` to the empty program. For this, an instance of Algorithm 3.1 was applied to several systems (other formalisms such as CCS, CSP, or the π-calculus can be tackled by providing different interpreters), followed by an abstract interpretation in the style of [29] (which we do not need in the present paper).

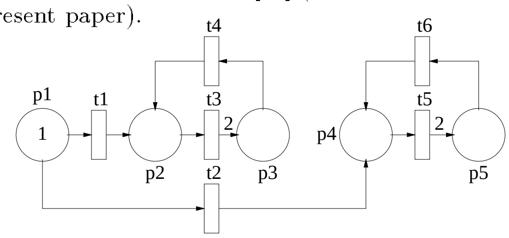


Figure 1: A simple Petri net

```
reachable(R) :-
       initial_marking(M), reachable(Tr,R,M).
reachable([],State,State).
reachable([Action|As],Reach,InState) :-
       trans(Action,InState,NewState),
       reachable(As,Reach,NewState).
initial_marking([s(0),0,0,0,0]).
trans(t1,[s(X),X2,X3,X4,X5],[X,s(X2),X3,X4,X5]).
trans(t2,[s(X),X2,X3,X4,X5],[X,X2,X3,s(X4),X5]).
trans(t3,[X,s(X2),X3,X4,X5],[X,X2,s(s(X3)),X4,X5]).
trans(t4,[X,X2,s(X3),X4,X5],[X,s(X2),X3,X4,X5]).
trans(t5,[X,X2,X3,s(X4),X5],[X,X2,X3,X4,s(s(X5))]).
trans(t6,[X,X2,X3,X4,s(X5)],[X,X2,X3,s(X4),X5]).
```

Figure 2: Encoding a Petri net as a logic program

We now show that this approach gives a decision procedure for the properties of Sect. 2 and that there is a tight link with existing Petri net algorithms.

We will first perform a preliminary compilation of the particular Petri net and task (Fig. 2). This will get rid of some of the interpretation overhead and also give us a more straightforward equivalence between markings of the Petri net and atoms encountered during the partial deduction phase proper. We will use the LOGEN offline partial deduction system [18] to that effect (but any other scheme which has a similar effect can be used). This system allows one to annotate calls in the original program as either reducible (executed by LOGEN) or non-reducible (not executed and thus kept in the specialised program).[9] In our case we will annotate all calls to `trans` and `initial_marking` as reducible. After that, the LOGEN system will (efficiently) produce a compiled version of Fig. 2: As can be seen in Fig. 3, the compilation gives us a predicate `reachable__1` with one argument each for the action label and the result, plus one argument per Petri net place. Observe that LOGEN (and ECCE as well) adds two underscores and a unique identifier to existing predicate names. `reachable__1` contains one

[8]When investigating the termination of Algorithm 3.2 we actually found a small mistake in the Karp_Miller_Tree Algorithm in [11], in the sense that it never terminates for unbounded Petri nets. We have presented a corrected version in this paper.

[9]LOGEN is offline: the control decisions have been taken beforehand (and are encoded in the annotations).

clause per transition of the Petri net plus one fact (for the marking reached). The initial marking is encoded in the one clause for `reachable__0`.

```
reachable__0(R) :- reachable__1(C,R,s(0),0,0,0,0).
reachable__1([],[B,C,D,E,F],B,C,D,E,F).
reachable__1([t1|A],R,s(B),C,D,E,F) :-
                  reachable__1(A,R,B,s(C),D,E,F).
reachable__1([t2|A],R,s(B),C,D,E,F) :-
                  reachable__1(A,R,B,C,D,s(E),F).
reachable__1([t3|A],R,B,s(C),D,E,F) :-
                  reachable__1(A,R,B,C,s(s(D)),E,F).
reachable__1([t4|A],R,B,C,s(D),E,F) :-
                  reachable__1(A,R,B,s(C),D,E,F).
reachable__1([t5|A],R,B,C,D,s(E),F) :-
                  reachable__1(A,R,B,C,D,E,s(s(F))).
reachable__1([t6|A],R,B,C,D,E,s(F)) :-
                  reachable__1(A,R,B,C,D,s(E),F).
```

Figure 3: Compiled Petri net

We now apply our partial deduction algorithms of Sect. 3 (using ECCE) to this compiled interpreter. The facts and rules of the resulting specialised program (see Appendix B) have the following form (apart from the top-level rule): Facts are of the form `reachable__1__N(...)` where `N` is a number (generated by ECCE) and the last parameter is a representation of a pseudo-marking. Rules are of the form `reachable__1__N1(...) :- reachable__1__N2(...)` where `N1`,`N2` are numbers and the first parameter of `reachable__1__N1(...)` contains a name of a transition. In Fig. 4 the facts generated from the code in Fig. 3 are represented as nodes and the generated rules as edges between these nodes. (The term `r_1_N` is a shortcut for `reachable_1_N`.)

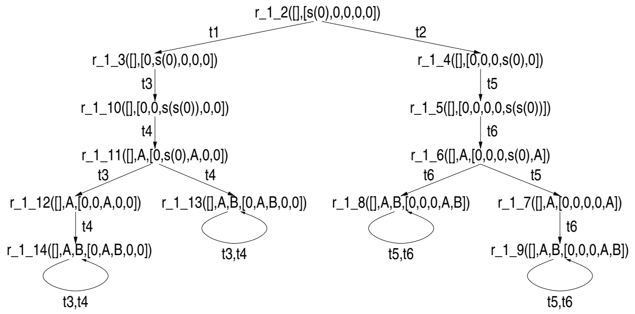
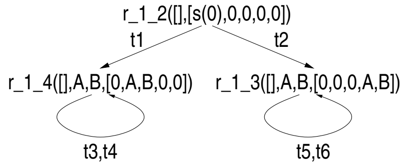


Figure 4: The result of applying ECCE to Fig. 3.

This graph bears a striking resemblance to the Karp–Miller tree for the above Petri net as given in [11]. In particular, all coverability problems which are decidable using the Karp–Miller tree can also be decided using the graph of Fig. 4. For example, from the existence of a fact for `r_1_13` with unbound variables for places 2 and 3 we can actually deduce that the places $p2$ and $p3$ are not bounded.

In the next section we prove that the relation between the code generated by Algorithm 3.2 and the Karp–Miller tree is not a coincidence. Furthermore, partial deduction can also be used to generate minimal coverability sets: The graph below was generated by Algorithm 3.3 applied to the example. Each node corresponds to one element of the minimal coverability set.



Using more natural (and powerful) settings of Algorithm 3.1, we can also handle problems which cannot be handled by any algorithm in [11]. For example, in [26], we applied Algorithm 3.1 to the manufacturing system used in [3] and were able to prove absence of deadlocks for parameter values of e.g., 1,2,3. When leaving the parameter unspecified, the system was unable to prove the absence of deadlocks and produced a residual program with facts. And indeed, for parameter value ≥ 9 the system can actually deadlock. The timings seem to compare favourably with HyTech [15].

5. COMPLETENESS RESULTS

We will now formally prove that the algorithms from Sect. 3 can be used to decide the coverability problems of Sect. 2. For this we need to establish a link between pseudo-markings from within the coverability sets of Section 2 and the atoms produced by our partial deduction algorithms.

First, we will denote by $C(\Pi, M_0)$ the logic program obtained by applying LOGEN to (a variation of) Fig. 2 encoding the Petri net Π with the initial marking M_0. The encoding of natural numbers as terms used by $C(\Pi, M_0)$ is:

- $\lceil i \rceil = 0$ if $i = 0$
- $\lceil i \rceil = s(\lceil i-1 \rceil)$ otherwise

Now, queries and goals for $C(\Pi, M_0)$ may also contain variables, which actually enables us to mimic the ω from Sect. 2. Hence, we extend our notation $\lceil . \rceil$ to $I\!N \cup \{\omega\}$ by defining $\lceil \omega \rceil = X$, where X is a fresh variable.

From now on, we suppose that the order of the places in M_0 is the same as in the encoding in Fig. 2. We can now establish a precise relationship between firing sequences of a Petri net and SLD-derivations of the above logic program translation:

LEMMA 1. *Let Π be Petri Net, and let M_0 and $M_k = \langle m_1, \ldots, m_n \rangle$ be two (pseudo) markings for Π.*
Then $M_0[t_1, \ldots, t_k\rangle M_k$ iff there exists an SLD-refutation for $C(\Pi, M_0) \cup \{\leftarrow reachable\_\_0(X)\}$ with computed answer $\{X/[\lceil m_1 \rceil, \ldots, \lceil m_n \rceil]\}$.
Also, $M_0[t_1, \ldots, t_k\rangle M_k$ iff there exists an SLD-derivation for $C(\Pi, M_0) \cup \{\leftarrow reachable\_\_0(X)\}$ leading to (a variant of) $\leftarrow reachable\_\_1(T, X, \lceil m_1 \rceil, \ldots, \lceil m_n \rceil)$. Furthermore, the refutation and the derivation will both always be of length $k+1$.

Next, recall that an atom for partial deduction denotes that all its instances. So, if during partial deduction we encounter `reachable_1(T,R,M1,...,Mk)` this means that any (ordinary) marking in $\{\langle m_1, ..., m_k \rangle \mid \exists\theta\forall i : \mathtt{Mi}\theta = \lceil m_i \rceil\}$ is (potentially) reachable. In particular, if any `Mi` is an unbound variable X this denotes that all values are potentially possible for that place. However, partial deduction atoms are more expressive than pseudo-markings: e.g., we can represent all $\langle m_1, m_2 \rangle$ such that $m_1 > 0$ and $m_2 = m_1 + 1$ by `reachable_1(T,R,s(X),s(s(X)))`. In other words, we can establish a link between the number of tokens in several places via shared variables and we can represent minimum values for places.[10] However, this information will be thrown away by *nmsg* and *naive_partition* (but not by partial deduction in general).

For a pseudo-marking $M = \langle m_1, \ldots, m_k \rangle$ we can now define $\lceil M \rceil_{p(\bar{t})} = p(\bar{t}, \lceil m_1 \rceil, \ldots, \lceil m_k \rceil)$ (where the $\bar{t}$ are just some extra arguments not directly related to the marking; in the previous section the encodings contained one extra argument registering the firing trace and one for the result). For example, we have $\lceil \langle 0, \omega, 2, \omega \rangle \rceil_{p(t)} = p(t, 0, X, s(s(0)), Y)$. Finally, as $\lceil . \rceil$ is injective we can define the (partial)[11] inverse $\lceil . \rceil^{-1}$.

THEOREM 1. *Let γ be produced by applying to $C(\Pi, M_0)$ and reachable__0(X) any instance of Algorithm 3.1 which uses naive_partition, a one-step unfolding rule, and where $whistle(L, \gamma) = A \Rightarrow A \trianglelefteq^- L$, and $abstract(L, W, \gamma) = nmsg(\{label(M_1), \ldots, label(M_k)\})$ for some ancestors M_i of L with $label(M_i) \trianglelefteq^- label(L)$. Then $\{\lceil label(A) \rceil^{-1} \mid A \in \gamma\}$ is a coverability set.*

The above theorem establishes that Algorithms 3.3 and 3.2 or any other valid instance of Algorithm 3.1(e.g., using more powerful whistles based upon $\trianglelefteq$ and characteristic trees), can be used as a decision procedure for the finite reachability set problem, the boundedness problem, the covering problem, and the quasi-liveness problem (see Sect. 2). For example, to decide whether $RS(\Pi)$ is infinite we report *true* as soon as the *whistle* blows; if the algorithm completes without any whistle blowing we report *false*.

For the quasi-liveness and the covering problem we can make the decision more explicit. For instance, to decide whether a a transition $t \in T$ of a Petri net $\Pi = (S, T, F, M_0)$ is quasi–live, we simply adapt the initial code from Fig. 2:

```
reachable([],State,State) :- trans(t,State,_).
```

and mark this `trans` call as reducible for LOGEN. We then run the same process as above, and if the final specialised program contains a fact then we report true, otherwise we report false. Note that in the latter case, the bottom-up abstract interpretation [29] used in [14, 26] will produce the empty program. We have thus established a class of infinite state systems and properties for which the approach of [14, 26] arrives at a definite, precise result. We now go even further and establish a 1-1 correspondence to the Karp-Miller procedure:

THEOREM 2. *Algorithm 3.2 applied to $C(\Pi, M_0)$ and reachable__0(X) will produce a global tree γ which is isomorphic to a Karp-Miller coverability tree of Π for the initial marking M_0.*

As we have seen earlier on one example, Algorithm 3.2 does indeed reproduce exactly the same result as the one presented in [11].

Now, while Algorithm 3.2 mimics the Karp-Miller procedure, Algorithm 3.3 was designed to mimic Finkel's minimal coverability algorithm [11]. Indeed, for the example in Sect. 4 it actually does so (cf. Appendix C). Unfortunately, this is not always the case. The problem is that Finkel's algorithm "drops" a marking (e.g., $\langle 0 \rangle$) if there is a bigger marking (e.g., $\langle 1 \rangle$) somewhere else in the tree. In the case of partial deduction, this "optimisation" corresponds to not specialising a call $p(0)$ if we have already specialised $p(s(0))$. Unfortunately, for logic programs in general this is unsound, and hence is not performed by partial deduction! Take for example the program:

$$p(0) \leftarrow true$$
$$p(s(0)) \leftarrow false$$

Here $\leftarrow p(0)$ succeeds while $\leftarrow p(s(0))$ fails and the above "optimisation" would lose a computed answer and produce an incorrect specialised program. For Petri nets, however, such an optimisation is correct due to the following monotonicity property: if $M[t_1, \ldots, t_k\rangle M'$ and $M'' \rangle M$ then $M''[t_1, \ldots, t_k\rangle M'''$ for some $M''' > M'$. This is reflected in the logic program encoding (cf. Fig. 2 and its compilation). Indeed, $M > M'$ implies that whenever $\lceil M' \rceil$ unifies with a clause, so does $\lceil M \rceil$. Syntactically, this corresponds to the fact that the pre-condition for firing a transition for a place is either a variable X or a term of the form $s(\ldots s(X) \ldots)$ (where X is not shared with another place). This insight leads to the following definition:

Definition 7. Let $\preceq$ be a quasi-order on atoms and let P be a logic program. Then $\preceq$ is a *valid covering relation* for P iff $A \preceq B$ implies that whenever A unifies via the *mgu* θ with the head H of a clause $H \leftarrow A_1, \ldots, A_m$ of P then

- B also unifies with H
- for some *mgu* σ of B and H we have that $\forall j \in \{1, \ldots, m\}$: $A_j\theta \preceq A_j\sigma$.

The above ensures that if we analyse B we will also cover (in the sense of $\preceq$) all computations that A can perform. The relations "instance of" and "variant of" are valid covering relations for any program P. However, for a particular program or class of programs stronger covering relations can be used.

This means that we can use a valid covering relation $\preceq$ within Algorithm 3.1 and still ensure correctness of our analysis (and correctness of the residual program if we adapt the renaming function). This is a very special instance of the abstract partial deduction framework in [22] and is actually

[10] More precisely, each atom represents a linear set $L \subseteq I\!N^k$ of markings $L = \{b + \sum_{i=1}^{r} n_i p^i \mid n_i \in I\!N\}$ with $b, p^i \in I\!N^k$ and the restriction that $\sum_{i=1}^{r} p^i \leq \langle 1, \ldots, 1 \rangle$.

[11] As mentioned above, all pseudo-markings have an encoding as an atom but not all atoms (generated by arbitrary instances of Algorithm 3.1) have an equivalent pseudo-marking.

what we need to fully mimic Finkel's minimal coverability algorithm by using $\preceq_{\lceil.\rceil}$ for the *covered*-test where: $\lceil m \rceil_{p(\bar{t})} \preceq \lceil m' \rceil_{p(\bar{t})}$ iff $m \leq m'$. $\preceq_{\lceil.\rceil}$ is a valid covering relation for programs of the form $C(\Pi, M_0)$ (but not for programs in general).

THEOREM 3. *Algorithm 3.3 using* $\preceq_{\lceil.\rceil}$ *for covered applied to* $P = C(\Pi, M_0)$ *and* $Q = reachable\_\_0(X)$ *will construct the minimal coverability set of* Π *for the initial marking* M_0.

6. FUTURE WORK

One big advantage of the partial deduction approach to model checking is it scales up to any formalism expressible as a logic program. More precisely, proper instantiations of Algorithm 3.1 will terminate for any system and will provide safe approximations of properties under consideration. However, as is to be expected, we might no longer have a decision procedure.

First, the partial deduction can handle a rather straightforward extension of [19, 11]. We can prove properties for infinite numbers of Petri nets by allowing ω's to be put into the initial markings. This was successfully used in [26] to prove a mutual exclusion safety property for an infinite family of systems. [26] discusses how to extend the model checking approach to liveness properties and full CTL. Some simple examples are solved. Reachability can be decided in some but not all cases using the present algorithms. In future we want to examine the relationship to Mayr's algorithm [30] and whether it can be mimicked by abstract partial deduction [22].

There are many extensions of the basic Petri net model. It turns out that most of these can be handled very easily within our approach by simple extensions to our interpreter. Boundedness is undecidable for Petri nets with reset arcs [8]. However, coverability and quasi-liveness is still decidable using a backwards algorithm [1, 12]. It turns out that our generic partial deduction algorithm can mimic this algorithm as well and thus provides a *decision procedure for the coverability problem of Petri nets with reset arcs*! For this we have to write an "inverse" interpreter and analyse it using partial deduction (but using *msg* instead of *nmsg* and not using *naive_partition*). In the companion paper [24] we study such Petri net extensions as well as the class of so-called well-structured transition systems. In future work we aim to analyse our approach in the context of other formalisms such as process algebras (first experiments with the π-calculus have been performed).

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. One can draw, however, a quite interesting conclusion about the worst-case complexity of some existing program specialisation techniques. Indeed, it should not be very difficult to adapt the arguments in this paper to show that not only partial deduction based upon $\trianglelefteq$ [25] but also supercompilation with $\trianglelefteq$ [37] or partial evaluation of functional-logic programs with $\trianglelefteq$ [2] can be used to decide the boundedness problem of Petri nets. Now, as the boundedness problem is known to be exponential-space hard [27], this means that above techniques should better not be used as is in a context (such as within a compiler) where a tight upper-bound on memory and time requirements is essential.

Another future direction is to move from pure logic programming to constraint logic programming. The latter allows for a more natural encoding of time (and arguably of Petri net markings) and has already proven to be useful on its own (i.e., without specialisation or analysis) for model checking [6, 13].

Acknowledgements

We would like to thank Javier Esparza, Neil Jones, Thierry Massart, and Ulrich Ultes-Nitsche for interesting discussions and comments. We also thank the referees for their detailed feedback.

7. REFERENCES

[1] P. A. Abdulla, K. Čerāns, B. Jonsson, and Y.-K. Tsay. General decidability theorems for infinite-state systems. In *Proceedings of LICS'96*, pages 313–321, July 1996. IEEE Computer Society Press.

[2] M. Alpuente, M. Falaschi, G. Vidal. Partial Evaluation of Functional Logic Programs. *ACM Transactions on Programming Languages and Systems*, 20(4):768–844, 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.

[4] W. Charatonik and A. Podelski. Set-based analysis of reactive infinite-state systems. In B. Steffen, editor, *Proceedings of TACAS'99*, LNCS 1384, pages 358–375. Springer-Verlag, March 1998.

[5] 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, November 1999.

[6] G. Delzanno and A. Podelski. Model checking in CLP. In R. Cleaveland, editor, *Proceedings of TACAS'99*, LNCS 1579, pages 223–239. Springer-Verlag, 1999.

[7] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pages 243–320. Elsevier, MIT Press, 1990.

[8] C. Dufourd, A. Finkel, and P. Schnoebelen. Reset nets between decidability and undecidability. In *Proceedings of ICALP'98*, LNCS 1443, pages 103–115. Springer-Verlag, 1998.

[9] E. A. Emerson and K. S. Namjoshi. On model checking for nondeterministic infinite state systems. In *Proceedings of LICS'98*, pages 70–80, 1998.

[10] J. Ezparza. Decidability of model-checking for infinite-state concurrent systems. *Acta Informatica*, 34:85–107, 1997.

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

[12] 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.

[13] L. Fribourg. Constraint logic programming applied to model checking. In A. Bossi, editor, *Proceedings of LOPSTR'99*, LNCS 1817, pages 31–42, Venice, Italy, September 1999.

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

[16] G. Higman. Ordering by divisibility in abstract algebras. *Proceedings of the London Mathematical Society*, 2:326–336, 1952.

[17] N. D. Jones. An introduction to partial evaluation. *ACM Computing Surveys*, 28(3):480–503, September 1996.

[18] 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.

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

[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] M. Leuschel. The ECCE partial deduction system and the DPPD library of benchmarks. Obtainable via `http://www.cs.kuleuven.ac.be/~dtai`, 1996.

[22] 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.

[23] M. Leuschel. Logic program specialisation. In J. Hatcliff, T. Æ. Mogensen, and P. Thiemann, editors, *Partial Evaluation: Practice and Theory*, LNCS 1706, pages 155–188, 1999. Springer-Verlag.

[24] M. Leuschel and H. Lehmann. Coverability of reset Petri nets and other well-structured transition systems by partial deduction. In J. Lloyd, editor, *Proceedings of CL'2000*, LNCS, 2000. Springer-Verlag. to appear.

[25] 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.

[26] M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialisation. In A. Bossi, editor, *Proceedings of LOPSTR'99*, LNCS 1817, pages 63–82, Venice, Italy, September 1999.

[27] R. Lipton. The reachability problem and the boundedness problem for Petri nets are exponential-space hard. *Conference on Petri Nets and Related Methods*, M.I.T., July 1975.

[28] J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11(3& 4):217–242, 1991.

[29] K. Marriott, L. Naish, and J.-L. Lassez. Most specific logic programs. *Annals of Mathematics and Artificial Intelligence*, 1:303–338, 1990.

[30] E. W. Mayr. An algorithm for the general Petri net reachability problem. *Siam Journal on Computing*, 13:441–460, 1984.

[31] F. Moller. Infinite results. In *Proceedings of CONCUR'96*, LNCS 1119, pages 195–216. Springer-Verlag, 1996.

[32] C. Rackoff. The covering and boundedness problems for vector addition systems. *Theoretical Computer Science*, 6(2):223–231, Apr. 1978.

[33] 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 *Proceedings of CAV'97*, LNCS 1254, pages 143–154. Springer-Verlag, 1997.

[34] W. Reisig. *Petri Nets - An Introduction.* Springer Verlag, 1982.

[35] J. Rushby. Mechanized formal methods: Where next? In *Proceedings of FM'99*, LNCS 1708, pages 48–51, Toulouse, France, Sept. 1999. Springer-Verlag.

[36] M. H. Sørensen. Convergence of program transformers in the metric space of trees. In *Proceedings of MPC'98*, LNCS 1422, pages 315–337. Springer-Verlag, 1998.

[37] 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.

[38] R. Valk and G. Vidal-Naquet. Petri nets and regular languages. *Journal of Computer and System Sciences*, 23(3):299–325, Dec. 1981.

[39] P. Wolper and B. Boigelot. Verifying systems with infinite but regular state spaces. In *Proceedings of CAV'98*, LNCS, pages 88–97. Springer-Verlag, 1998.

APPENDIX

A. PROOFS

LEMMA 2. *For $i, j \in I\!N \cup \{\omega\}$ we have that $i \leq j$ iff $\lceil i \rceil \trianglelefteq^- \lceil j \rceil$. Similarly, for two pseudo-markings M, M' and two atoms A, B such that $A \trianglelefteq^- B$ we have that $M \leq M'$ iff $\lceil M \rceil_A \trianglelefteq^- \lceil M' \rceil_B$.*

LEMMA 3. *Given two pseudo-markings M, M' define the pseudo-marking M'' by: $M''(p) = M(p)$ if $M(p) = M'(p)$ and $M''(p) = \omega$ otherwise.*
Then $\lceil M'' \rceil_{p(\bar{t})} = nmsg(\lceil M \rceil_{p(\bar{t})}, \lceil M' \rceil_{p(\bar{t})})$.

We from now on write $s \approx \lceil M \rceil$ if $\lceil M \rceil_{p(\bar{t})} = s$ for some atom $p(\bar{t})$.

THEOREM 1 Let γ be produced by applying to $C(\Pi, M_0)$ and *reachable__0*(X) any instance of Algorithm 3.1 which uses *naive_partition*, a one-step unfolding rule, and where $whistle(L, \gamma) = A \Rightarrow A \trianglelefteq^- L$, and $abstract(L, W, \gamma) = nmsg(\{label(M_1), \ldots, label(M_k\})$ for some ancestors M_i of L with $label(M_i) \trianglelefteq^- label(L)$. Then $\{\lceil label(A) \rceil^{-1} \mid A \in \gamma\}$ is a coverability set.

PROOF. *(Sketch)* First, it can easily be proven by induction that all the labels of any γ arising during the execution of the algorithm can be associated with a pseudo-marking of P in the sense that $L \in \gamma \Rightarrow label(L) \approx \lceil M \rceil$. (Obviously this holds for the initial tree γ and will not be violated by *abstract* as it uses *nmsg* nor by adding children due to correctness of the unfolding step [28] and because *naive_partition* will replace any non-ground term by a fresh variable corresponding to ω.)

The first condition for being a coverability set (Definition 2) is that all markings of the reachability set are covered. This is a straightforward consequence of a) the fact that termination of our algorithm implies closedness and thus also correctness of partial deduction [28], b) the fact the we use a 1-step unfolding rule (and hence no intermediate markings can be hidden) and c) of the correctness of our interpreter (Lemma 1).

The second condition is a consequence of the following: let $M = \lceil label(A) \rceil^{-1}$ (which is defined by our first point above) with $A \in \gamma$. If M has no ω then M itself is a reachable marking as a) our interpreter (Lemma 1) is correct, b) partial deduction is sound and c) in this case also complete (as no abstraction has occurred to reach M). Indeed, for c):

- the only abstraction performed by *nmsg* and *naive_partition* is to introduce ω;
- once an ω has been introduced it can never disappear,[12]
- concerning *partition* all leaf goals are atoms so there is no abstraction required to split the leaf goals into individual atoms).

If M contains an ω then we know that a fresh variable was at some point introduced by *nmsg* within a call to $abstract(L, \gamma)$ (if *partition* puts in an ω then an ω was already present before). By Definition 5 we know that $abstract(L, \gamma) = nmsg(\ldots (nmsg(label(L), label(M_1)) \ldots), label(M_k))$ where $M_1, \ldots, M_k$ are ancestors of L (not necessarily all) with $label(M_i) \trianglelefteq^- label(L)$. By Lemma 2 we know that $\lceil label(M_i) \rceil^{-1} \leq \lceil label(L) \rceil^{-1}$. We also know that for an ω to be introduced we must have for some i's (and we can actually ignore the others without changing the result of *abstract*) that $label(M_i)$ and $label(L)$ are not variants. This implies that for those i's: $\lceil label(M_i) \rceil^{-1} < \lceil label(L) \rceil^{-1}$. Hence by the monotonicity of Petri nets[13] we can construct an infinite sequence of transitions whose limit is the generalisation M. □

THEOREM 2 Algorithm 3.2 applied to $P = C(\Pi, M_0)$ and $Q =$ *reachable__0*(X) will produce a global tree γ which is isomorphic to a Karp-Miller coverability tree of Π for the initial marking M_0.

PROOF. *(Sketch)* We have (see proof of Theorem 1) that $L \in \gamma \Rightarrow label(L) \approx \lceil M \rceil$.

Let us now examine all the cases of the Karp-Miller algorithm and prove that they are mimicked by the partial deduction algorithm.

Case 1 is mimicked by the *covered* test as $M = M_1$ implies that $\lceil M \rceil_{p(\bar{t})}$ is a variant of $\lceil M_1 \rceil_{p(\bar{t})}$.

If case 2 applies, we have by Lemma 2 that the *whistle* blows (the extra argument 1 is a fresh variable, and the extra argument 2 is always the same [and usually a fresh variable as well], which implies that the condition $A \trianglelefteq^- B$ of Lemma 2 is satisfied). We also know by the same lemma that *covered* does not hold. Let us first assume that there is only one ancestor (k_1, M_1). In that case we can directly apply Lemma 3 to show that for the new label A computed by *abstract* we have exactly $A \approx \lceil M_2 \rceil$. In case there are more than one ancestor (k_1, M_1) the same can be proven using a straightforward induction on the number of such ancestors.

If case 3 applies we know by Lemma 2 that neither *covered* nor *whistle* are true in the partial deduction algorithm. For markings without ω, we know by correctness of unfolding and Lemma 1 we know that the partial deduction algorithm will add exactly the same children as the Karp-Miller procedure (and will also label them using a clause corresponding to the same transition). For markings with ω we also get the same children as with the Karp-Miller procedure because *naive_partition* will replace any non-ground term by a fresh variable corresponding to ω. □

THEOREM 3 Algorithm 3.3 using $\preceq_{\lceil . \rceil}$ for *covered* applied to $P = C(\Pi, M_0)$ and $Q =$ *reachable__0*(X) will construct the minimal coverability set of Π for the initial marking M_0.

PROOF. *(Sketch)* We can show that Algorithm 3.3 mimics all the 4 cases of "minimal_coverability_tree" procedure in [11] for a currently selected node with marking m. Let L be the node selected in Algorithm 3.3 corresponding to m, i.e., $\lceil label(L) \rceil = m$ and let us examine the 4 cases:

1. if there is a node with marking $m1 = m$ then $covered(L, \gamma) = true$ and the node L will thus be marked as processed
2. if there is a node with marking $m1 < m$ then $covered(L, \gamma) = true$ and the node L will thus be marked as processed. Here it is important that *covered* is extended using $\preceq_{\lceil . \rceil}$.
3. let us first consider the case that there is an ancestor of m such that $m_1 < m$. In that case the $whistle(L, \gamma)$ will blow (by Lemma 2) and return the first node on the path from the root such that $m_1 < m$ ($m_1 = m$ is not possible as case 1 hasn't been applied) By Lemma 3 $abstract(L, W, \gamma)$ we now know that $abstract(L, W, \gamma)$ will give exactly m_2

[12] For Petri nets with reset arcs this is no longer the case.

[13] I.e., $M[t_1, \ldots, t_k\rangle M'$ and $M''\rangle M$ implies that $M''[t_1, \ldots, t_k\rangle M'''$ for some $M'''\rangle M'$.

of the "minimal_coverability_tree" procedure. Also, the removal of all nodes such that $m_1 < m_2$ will be performed by the post-processing of Algorithm 3.3 (if the with m_2 got removed then there will be another node with a marking $> m_2$ which will remove all the nodes with a marking $< m_2$ as well.
For the case that there is no ancestor of m such that $m_1 < m$ we have that $m_2 = m$. Again, the removal of all nodes such that $m_1 < m_2$ will be performed by the post-processing of Algorithm 3.3. In this case the $whistle(L, \gamma)$ of Algorithm 3.3 does not blow but at the next iteration of the "minimal_coverability_tree" procedure m_2 will fall into case 4.

4. In this case we know that the $whistle(L, \gamma)$ does not blow (by Lemma 2) and similarly to Theorem 2 we can establish that the partial deduction unfolding exactly mimics the construction of the children in [11].

□

B. A KARP-MILLER TREE USING ECCE

In this appendix we show how ECCE reconstructs precisely the Karp-Miller tree for the Petri net PN1 of [11] (page 219).

```
/* Specialised program generated by Ecce 1.1 */
/* PD Goal: reachable__0(A) */
/* Parameters: Abs:n InstCheck:y Msv:n NgSlv:g Part:n
   Prun:n Sel:c Whstl:o Raf:noFar:no Dce:no Poly:n
   Dpu:no ParAbs:no Msvp:no */
/* Transformation time: 134 ms */
/* Specialised Predicates:
reachable__0__1(A) :- reachable__0(A).
reachable__1__2(A,B) :- reachable__1(A,s(0),0,0,0,0,B).
reachable__1__3(A,B) :- reachable__1(A,0,s(0),0,0,0,B).
reachable__1__4(A,B) :- reachable__1(A,0,0,0,s(0),0,B).
reachable__1__5(A,B) :- reachable__1(A,0,0,0,0,s(s(0)),B).
reachable__1__6(A,B,C) :- reachable__1(A,0,0,0,s(0),B,C).
reachable__1__7(A,B,C) :- reachable__1(A,0,0,0,0,B,C).
reachable__1__8(A,B,C,D) :- reachable__1(A,0,0,0,B,C,D).
reachable__1__9(A,B,C,D) :- reachable__1(A,0,0,0,B,C,D).
reachable__1__10(A,B) :- reachable__1(A,0,0,s(s(0)),0,0,B).
reachable__1__11(A,B,C) :- reachable__1(A,0,s(0),B,0,0,C).
reachable__1__12(A,B,C) :- reachable__1(A,0,0,B,0,0,C).
reachable__1__13(A,B,C,D) :- reachable__1(A,0,B,C,0,0,D).
reachable__1__14(A,B,C,D) :- reachable__1(A,0,B,C,0,0,D).*/

reachable__0(A) :- reachable__0__1(A).

reachable__0__1(A) :- reachable__1__2(B,A).
reachable__1__2([],[s(0),0,0,0,0]).
reachable__1__2([t1|A],B) :- reachable__1__3(A,B).
reachable__1__2([t2|A],B) :- reachable__1__4(A,B).
reachable__1__3([],[0,s(0),0,0,0]).
reachable__1__3([t3|A],B) :- reachable__1__10(A,B).
reachable__1__4([],[0,0,0,s(0),0]).
reachable__1__4([t5|A],B) :- reachable__1__5(A,B).
reachable__1__5([],[0,0,0,0,s(s(0))]).
reachable__1__5([t6|A],B) :- reachable__1__6(A,s(0),B).
reachable__1__6([],A,[0,0,0,s(0),A]).
reachable__1__6([t5|A],B,C) :-
          reachable__1__7(A,s(s(B)),C).
reachable__1__6([t6|A],s(B),C) :-
          reachable__1__8(A,s(s(0)),B,C).
reachable__1__7([],A,[0,0,0,0,A]).
reachable__1__7([t6|A],s(B),C) :-
          reachable__1__9(A,s(0),B,C).
reachable__1__8([],A,B,[0,0,0,A,B]).
reachable__1__8([t5|A],s(B),C,D) :-
          reachable__1__8(A,B,s(s(C)),D).
reachable__1__8([t6|A],B,s(C),D) :-
          reachable__1__8(A,s(B),C,D).
reachable__1__9([],A,B,[0,0,0,A,B]).
reachable__1__9([t5|A],s(B),C,D) :-
          reachable__1__9(A,B,s(s(C)),D).
reachable__1__9([t6|A],B,s(C),D) :-
          reachable__1__9(A,s(B),C,D).
reachable__1__10([],[0,0,s(s(0)),0,0]).
reachable__1__10([t4|A],B) :-
          reachable__1__11(A,s(0),B).
reachable__1__11([],A,[0,s(0),A,0,0]).
reachable__1__11([t3|A],B,C) :-
          reachable__1__12(A,s(s(B)),C).
reachable__1__11([t4|A],s(B),C) :-
          reachable__1__13(A,s(s(0)),B,C).
reachable__1__12([],A,[0,0,A,0,0]).
reachable__1__12([t4|A],s(B),C) :-
          reachable__1__14(A,s(0),B,C).
reachable__1__13([],A,B,[0,A,B,0,0]).
reachable__1__13([t3|A],s(B),C,D) :-
          reachable__1__13(A,B,s(s(C)),D).
reachable__1__13([t4|A],B,s(C),D) :-
          reachable__1__13(A,s(B),C,D).
reachable__1__14([],A,B,[0,A,B,0,0]).
reachable__1__14([t3|A],s(B),C,D) :-
          reachable__1__14(A,B,s(s(C)),D).
reachable__1__14([t4|A],B,s(C),D) :-
          reachable__1__14(A,s(B),C,D).
```

C. A MINIMAL COVERABILITY GRAPH USING ECCE

In this appendix we show how ECCE reconstructs precisely the minimal coverability graph (pages 223 and 230 of [11]) for the Petri net PN1 of [11].

```
/* Specialised program generated by Ecce 1.1 */
/* PD Goal: reachable__0(A) */
/* Parameters: Abs:n InstCheck:a Msv:n NgSlv:g
   Part:n Prun:n Sel:c Whstl:n Raf:noFar:no Dce:no
   Poly:n Dpu:no ParAbs:yes Msvp:no */
/* Transformation time: 83 ms */
/* Specialised Predicates:
reachable__0__1(A) :- reachable__0(A).
reachable__1__2(A,B) :-
          reachable__1(A,s(0),0,0,0,0,B).
reachable__1__3(A,B,C,D) :-
          reachable__1(A,0,0,0,B,C,D).
reachable__1__4(A,B,C,D) :-
          reachable__1(A,0,B,C,0,0,D). */

reachable__0(A) :- reachable__0__1(A).

reachable__0__1(A) :- reachable__1__2(B,A).

reachable__1__2([],[s(0),0,0,0,0]).
reachable__1__2([t1|A],B) :-
          reachable__1__4(A,s(0),0,B).
reachable__1__2([t2|A],B) :-
          reachable__1__3(A,s(0),0,B).

reachable__1__3([],A,B,[0,0,0,A,B]).
reachable__1__3([t5|A],s(B),C,D) :-
          reachable__1__3(A,B,s(s(C)),D).
reachable__1__3([t6|A],B,s(C),D) :-
          reachable__1__3(A,s(B),C,D).

reachable__1__4([],A,B,[0,A,B,0,0]).
reachable__1__4([t3|A],s(B),C,D) :-
          reachable__1__4(A,B,s(s(C)),D).
reachable__1__4([t4|A],B,s(C),D) :-
          reachable__1__4(A,s(B),C,D).
```