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, hel99}@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 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 nondeterminism) and enables us to express infinite state systems. First successful steps in that direction have been taken in [14, 25], 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. 9006882.

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 Finieli [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 $\Pi = (S, T, F, M_0)$ 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 $\Pi$ is a mapping $m : S \mapsto \mathbb{N}$. $M_0$ 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 $M[t]M'$. By $M[t_1, \ldots, t_k]M'$ we denote the fact that for some intermediate markings $M_1, \ldots, M_{k-1}$ we have $M[t_1]M_1, \ldots, M_{k-1}[t_k]M'$. We define the reachability tree $RT(\Pi)$ inductively as follows:

1. 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 $M[t]M'$ and add an arc from $n$ to $n'$ labelled $t$. The set of all labels of $RT(\Pi)$ is called the reachability set of $\Pi$, 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 $\Pi$, 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 $\omega + n = \omega = n + \omega = \omega$. Using this we also extend the notation $M_0[\omega_1, \ldots, \omega_k]M'$ 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 $M_{k'}(s) = M'(s)$.

Many interesting properties of Petri nets can be investigated using the so-called Karp-Miller tree resulting from the algorithm 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 $\Pi$, 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 $\omega$ 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 $M_2 = \langle 3, 3, 2 \rangle$ 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 $\langle 1, \omega, \omega, \omega \rangle$ 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 $U := \{(\epsilon, M_0)\}$ 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 $(k_2, M_2)$ of $(k, M)$ such that $M_1 < M$ do

for all places $p \in S$ such that $M_2(p) < M(p)$ do $M_2(p) := \omega$;

$M := M_2$;
for every transition $t$ such that $M[\epsilon]M'$ do

create node $(k, M')$;
create arc labelled $t$ from $(k, M)$ to $(k', M')$;
$U := U \cup (k', M')$;

3This deviates slightly from the definition in [11], according to which the sequence $(0, 0), (1, 0), (1, 1), (2, 1), \ldots$ does not converge to $(\omega, \omega)$.

4The 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 $\Pi$ infinite? Procedure: $RT(\Pi)$ is infinite iff the label of some node of $KM(\Pi)$ contains an $\omega$, or, some node has an ancestor labelled by the same marking.

2. Is $RS(\Pi)$ of a Petri net $\Pi$ infinite? Procedure: $RS(\Pi)$ is infinite iff the label of some node of $KM(\Pi)$ contains an $\omega$.

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 $I(\Pi)$ of a Petri net $\Pi$ regular? Procedure: $I(\Pi)$ is regular iff for every node $n$ of $KM(\Pi)$ which is labelled by a marking containing $\omega$ and for every ancestor $n_\alpha$ of $n$ such that $n_\alpha$ is labelled by the same marking, we have that $\sum_{s \in S}(|s|)(F(t,s) - F(s,t)) \geq 0$ for all $s \in S$, where $\omega \in T^+$ is the path in $KM(\Pi)$ from $n_\alpha$ to $n$ and $|s|$ denotes the Parikh-mapping (i.e., $|s|$ is the number of $t$'s in $s$).

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 $\{M_\alpha\}$ converging to $M'$.

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

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 $P_\theta$ 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 $P_\theta$ should be correct for all runtime instantiations $\leftarrow Q'\theta$ of $\leftarrow Q'$ in the sense that the computed answers of $P_\theta \cup \{\leftarrow Q'\theta\}$ and $P_\theta \cup \{\leftarrow Q\theta\}$ 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 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 and possibly incomplete SLD-tree $\tau$ for $P \cup \{\leftarrow Q\}$. We also define the set of leaves, $\text{leaves}(\tau)$, to be the leaf goals of $\tau$.

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

4An 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.

A trivial SLD-tree has a single node where no literal has been selected for resolution.
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 naive (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.

3.2 Concrete Algorithms

If the Algorithm 3.1 terminates then the closedness condition of [28] is satisfied, i.e., it is ensured that together the SLD-trees γ1, . . . , γn form a complete description of all possible computations that can occur for all concrete instances ← A0 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 γ(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.

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, γ), a whistle function whistle(L, γ) and an abstraction function abstract(L, W, γ).

Intuitively, covered(L, γ) is a way of checking whether L or a generalisation of L has already been treated in the global tree γ. Formally, covered(L, γ) = true must imply that ∃M ∈ γ such that M is processed or abstracted and for some substitution θ: label(M)θ = 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 simply define partition(L) = {A1, . . . , An}. However, it is possible to perform some abstraction—indeed, the particular global tree γ at that stage. Formally, whenever partition(Q) = {A1, . . . , An} then for some θ1, . . . , θn we must have that Q is identical to ← Aθ1, . . . , Anθn (up to reordering of atoms).

The other two parameters are used to ensure termination. Intuitively, the whistle(L, γ) 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, γ) 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.
Abstraction

Putting it all together

Definition 4. A sequence $s_1,s_2,...$ of elements of $S$ is called admissible w.r.t 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 (wqfr) iff there are no infinite admissible sequences $\leq_S$. A well quasi order (wqo) is a reflexive and transitive wqfr.

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

Definition 5. Given a wqo $\leq$ we define $\text{whistle}_{\leq}$ as follows: $\text{whistle}_{\leq}(L,\gamma) = M$ iff $M$ is the farthest ancestor of $L$ such that $\text{label}(M) \leq \text{label}(L)$ and $\text{whistle}_{\leq}(L,\gamma) = \text{fail}$ 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 $\leq$ on expressions is inductively defined as follows (i.e. $\leq$ is the least relation satisfying the rules):

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

We can also use $\text{nmssg}$ to implement a naive partitioning function where $A' \in Q$ means that $A'$ is an atom in $Q$:

$\text{naive-partition}(Q) = \{A \mid A' \in Q \text{ and } A = \text{nmssg}(A',A')\}.

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

Due to their naïvety, neither $\text{nmssg}$ nor $\text{naive-partition}$ are used (to our knowledge) in existing online 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:

1. $\text{parent-abstract} = \text{false}$
2. $U$ unfolds just once.
3. $\text{partition}(Q) = \text{naive-partition}(Q)$
4. $\text{covered}(L,\gamma) = \text{true}$ if there exists an ancestor node of $L$ in $\gamma$ which is different from $L$ and whose label is a variant of $\text{label}(L)$
5. $\text{whistle}(L,\gamma) = \text{fail}$ if $L$ is marked as abstracted and $\text{whistle}(L,\gamma) = \text{whistle}_{\leq}(L,\gamma)$ otherwise
6. $\text{abstract}(L,W) = \text{nmssg}(\text{label}(M_1),\ldots,\text{label}(M_k))$ where $\{M_1,\ldots,M_k\}$ are all the ancestors of $L$ with $\text{label}(M_i) \leq \text{label}(L)$.

From a partial deduction perspective this algorithm is sub-optimal: $\text{covered}$ only looks at ancestors (normally one would look anywhere in the tree), $\text{partition}$ and $\text{nmssg}$ explicitly throw information away, the $\text{whistle}$ 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

$\text{partition}$ (as in)
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, γ) = true if there exists a processed node in γ whose label is more general than label(L)
- whistle(L, γ) = whistle(L, γ)
- in addition, after Algorithm 3.1 is complete, remove all nodes L (and their subtrees) for which covered(L, γ) = 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 ← 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 searchable for reachable markings in RS(Π). Other Petri Nets can be encoded by changing the trans/3 facts and the initialMarking 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(Π, State, State) which detects unsafe states. E.g., if we want to verify that there should never be tokens in the places p2 and p3 at the same time, we would write in Fig. 2:

```
reach(Π, 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 μ-calculus can be tackled by providing different interpreters), followed by an abstract interpretation in the style of [20] (which we do not need in the present paper).

![Figure 1: A simple Petri net](image1)

Reachable(Π) :- initialMarking(0), reachable(Tr, R, M).
Reachable(Π, State, State) :-
trans(Π, State, State),
reach(Π, State, State).
InitialMarking([s(0), 0, 0, 0, 0, 0], [X, S(X2), X3, X4, X5]).
Trans(t1, [s(0), X2, X3, X4, X5], [X, s(X2), X3, X4, X5]).
Trans(t2, [s(0), X2, X3, X4, X5], [X, X2, X3, s(X4), X5]).
Trans(t3, [X, s(X2), X3, X4, X5], [X, X2, X3, 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, s(X3), 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](image2)

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).² In our case we will annotate all calls to trans and initialMarking 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 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.

注①：LOGEN is offline: the control decisions have been taken beforehand (and are encoded in the annotations).

⁸When 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.
clause of transition of the Petri net plus one fact (for the marking reached). The initial marking is encoded in the one
clause for reachable_0.

\[
\text{reachable}_0(0) := \text{reachable}_1(C, R, s(0), 0, 0, 0).
\]

\[
\text{reachable}_1(i, [B, C, D, E, F], B, C, D, E, F) :=
\]

\[
\text{reachable}_1((i1[i], R, s(B), C, D, E, F) :=
\]

\[
\text{reachable}_1((i2[i], R, s(B), C, D, E, F) :=
\]

\[
\text{reachable}_1((i3[i], R, s(B), C, D, E, F) :=
\]

\[
\text{reachable}_1((i4[i], R, s(B), C, D, E, F) :=
\]

\[
\text{reachable}_1((i5[i], R, s(B), C, D, E, F) :=
\]

\[
\text{reachable}_1((i6[i], R, s(B), C, D, E, F) :=
\]

\[
\text{reachable}_1((i7[i], R, s(B), C, D, E, F)) :=
\]

\[
\text{reachable}_1((A, B, C, D, E, F, s(F))) :=
\]

\[
\text{reachable}_1((A, B, C, D, E, F, s(F))) :=
\]

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 reach-clause\(_i\) where \(i\) is a number (generated by ECCE) and the last parameter is a repre-
sentation of a pseudo-marking. Rules are of the form reach-clause\(_i\) \(\rightarrow\) reach-clause\(_{i'}\) where \(i, i'\) are
numbers and the first parameter of reach-clause\(_i\) 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
\(r_{ij}\) is a shortcut for reach-clause\(_i\) \(\rightarrow\) reach-clause\(_j\).)

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 \(r_{14}\) 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 ex-
ample. 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 [36], we apply Al-
gorithm 3.1 to the manufacturing system used in [3] and
were able to prove absence of deadlocks for parameter val-
ues 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 \(\geq 9\) 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 ob-
tained by applying LOGEN to (a variation of) Fig. 2 encoding
the Petri net \(\Pi\) with the initial marking \(M_0\). The encoding
of natural numbers as terms used by \(C(\Pi, M_0)\) is:

- \(\bar{Z} \equiv \bar{S} \equiv 0\) if \(i = 0\)
- \(\bar{Z} \equiv \bar{S}(\bar{Z} + 1)\) otherwise

Now, queries and goals for \(C(\Pi, M_0)\) may also contain vari-
ables, which actually enables us to mimic the \(\omega\) from Sect. 2.
Hence, we extend our notation \([\bar{Z}]\) to \(N \cup \{\omega\}\) by defining
\(\bar{Z} \equiv \bar{X}\) where \(\bar{X}\) 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 \(\Pi\) be Petri Net, and let \(M_0\) and \(M_k =
(m_1, \ldots, m_n)\) be two (pseudo) markings for \(\Pi\).
Then \(M_0([1, \ldots, t_k]M_0)\) iff there exists an SLD-refuta-
tion for \(C(\Pi, M_0)\) \(\langle \text{reach-clause}_0(X) \rangle\) with computed answer
\(\{X, [m_1], \ldots, [m_n]\}\).

Also, \(M_0([1, \ldots, t_k]M_0)\) iff there exists an SLD-ref-
tion for \(C(\Pi, M_0)\) \(\langle \text{reach-clause}_0(X) \rangle\) leading to (a variant of)
\(\langle \text{reach-clause}_i(T, X, [m_1], \ldots, [m_n]) \rangle\). 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
\(\text{reach-clause}_i(T, R, M_i)\) this means that any (ordi-
nary) marking in \(\{m_1, \ldots, m_n\}\) \(\exists M_0 : M_0 = [m_i]\) is (po-
tentially) reachable. In particular, if any \(M_i\) is an unbound
variable \(X\) this denotes that all values are potentially pos-
sible for that place. However, partial deduction atoms are
more expressive than pseudo-markings: e.g., we can represent all \((m_1, m_2)\) such that \(m_1 > 0\) and \(m_2 = m_1 + 1\) by reachable_{\gamma}(T, s(X)) in \(M(X, \gamma))\). 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. However, this information will be thrown away by \(msg\) and \(naive\)-\(partition\) (but not by partial deduction in general).

For a pseudo-marking \(M = (m_1, \ldots, m_n)\) we can now define \(M_{\gamma} = \gamma[M] = \gamma(V[M_1], \ldots, V[M_n])\) (where the \(V\) 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 \([0, \omega, 2, \omega])\) \(\in\) \(\gamma(\omega, \gamma, \omega))\). Finally, as \([\gamma]\) is injective we can define the (partial) \(^{-1}\) inverse \([\gamma]^{-1}\).

**Theorem 1.** Let \(\gamma\) be produced by applying to \(C(H, M_0)\) and \(reachable\)-\(D(X)\) any instance of Algorithm 3.1 which uses naive\-\(partition\), a one-step unfolding rule, and where \(\gamma\) is the 

whistle\-\(partition\) of the \(\alpha\) and \(\beta\) set of places, and \(\gamma\) is a covering relation of \(\alpha\). Then \(\gamma\) is a covering relation of \(\alpha\).

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 \(\leq\) 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 \(RS(H)\) is finite 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 \(\Pi\) is quasi-live, we simply adopt the initial code from Fig. 2:

\[
\text{reachable}(\Pi, State, State) := \text{trans}(t, State, \_).
\]

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 [28] 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 \(C(H, M_0)\) and \(reachable\)-\(D(X)\) will produce a global tree \(\gamma\) which is isomorphic to a Karp-Miller covering relation of \(\Pi\) 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 covering 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., \((0, 0)\)) if there is a bigger marking (e.g., \((1, 0)\)) 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 \(p(1)\) or \(p(2)\). Unfortunately, for logic programs in general this is unsound, and hence is not performed by partial deduction! Take for example the program:

\[
\begin{align*}
p(0) & \leftarrow true \quad p(0) \leftarrow false
\end{align*}
\]

Here \(\neg p(0)\) succeeds while \(\neg p(0)\) 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 \(M(t_1, \ldots, t_k) > M'\) and \(M'' > M\) then \(M''(t_1, \ldots, t_k) > M''\). This is reflected in the logic program encoding (cf. Fig. 2 and its compilation). Indeed, \(M > M'\) implies that whenever \(M'\) unifies with a clause, so does \(M\). 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 \(s(X)\) (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 with the \(\text{msg}\) head \(\Pi\) of a clause \(\Pi \leftarrow A_1, \ldots, A_m\) of \(P\) then:

- \(B\) also unifies with \(H\)
- for some \(\text{msg}\) \(\sigma\) of \(B\) and \(H\) we have that \(\forall j \in \{1, \ldots, m\} : A_j \preceq A_j \sigma\).

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 specialized program if we adapt the renaming function). This is a very special instance of the abstract partial deduction framework in [22] and is actually
what we need to fully mimic Finkel's minimal coverability algorithm by using \( \preceq \) for the closed-form where \( [m]_{P} \preceq [m']_{P} \) if \( m \preceq m' \). \( \preceq \) is a valid covering relation for programs of the form \( C(P, M) \) (but not for programs in general).

**Theorem 3.** Algorithm 3.3 using \( \preceq \) for cover applied to \( P = C(P, M) \) and \( Q = \text{reachable}(X) \) will construct the minimal coverability set of \( P \) for the initial marking \( M_0 \).

6. **FUTURE WORK**

One big advantage of the partial deduction approach to model checking is that 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 \( w \)'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. [25] 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 partial algorithms. In future work 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 approach [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 mmsg 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 \( \pi \)-calculus have been performed).

An important aspect of model checking for 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 \( \preceq \) but also supercomilation with \( \preceq \) [37] or partial evaluation of functional-logic programs with \( \preceq \) [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 Jeltsch for interesting discussions and comments. We also thank the referees for their detailed feedback.

7. **REFERENCES**


APPENDIX

A. PROOFS

Lemma 2. For i, j ∈ N ∪ {ω} we have that i ≤ j iff [i] ⊆* [j]. Similarly, for two pseudo-markings M, M' and two atoms A, B such that A ≤* B we have that M ≤ M' iff [M]A ≤* [M']A.

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) = ω otherwise.


We from now on write s ≈ [M] if [M]A = s for some atom p(A).

Theorem 1 Let γ be produced by applying to C(Π, M6) and reachable_ann(X) any instance of Algorithm 3.1 which uses naïve_partition, a one-step unfolding rule, and where whistle(L, γ) = A ⇒ A ≤* L and abstract(L, W, γ) = rmsg(label(M1),..., label(M3)) for some ancestors Mi of L with label(Mi) ≤* label(L). Then {label(A)}−1 | A ∈ γ is a covering 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 ∈ γ ⇒ label(L) ≈ [M]. (Obviously this holds for the initial tree γ and will not be violated by abstract as it uses rmsg nor by adding children due to correctness of the unfolding step [28] and because naïve_partition will replace any non-ground term by a fresh variable corresponding to ω.)

The first condition for being a covering set (Definition 2) is that all markings of the readability set are covered. This is a straightforward consequence of a) the fact that termination of our algorithm implies completeness 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 = label(A)−1 (which is defined by our first point above) with A ∈ γ. 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 rmsg and naïve_partition is to introduce ω;
- once an ω has been introduced it can never disappear;
- 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 rmsg within a call to abstract(L, γ) (if partition puts in an ω then an ω was already present before). By Definition 5 we know that abstract(L, γ) = rmsg(label(L), label(M3),..., label(Mi)) where M1,..., Mi are ancestors of L (not necessarily all) with label(Mi) ≤* label(L). By Lemma 2 we know that {label(Mi)}−1 ≈ [label(L)]−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(Mi) and label(L) are not variants.

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

This implies that for those ω's [label(Mi)]−1 < [label(L)]−1. Hence by the monotonicity of Petri nets we can construct an infinite sequence of transitions whose limit is the generalisation M.

Theorem 2 Algorithm 3.2 applied to P = C(Π, M6) and Q = reachable_ann(X) will produce a global tree γ which is isomorphic to a Karp-Miller coverability tree of Π for the initial marking M6.

Proof. (Sketch) We have (see proof of Theorem 1) that L ∈ γ ⇒ label(L) ≈ [M].

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 = M1 implies that [M]A = a variant of [M]A.

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 ≤* 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 (k1, M1). In that case we can directly apply Lemma 3 to show that for the new label L computed by abstract we have exactly A ≈ [M2].

In case there are more than one ancestor (k1, M1) 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 naïve_partition will replace any non-ground term by a fresh variable corresponding to ω.

Theorem 3 Algorithm 3.3 using ≼* γ for covered applied to P = C(Π, M6) and Q = reachable_ann(X) will construct the minimal coverability set of Π for the initial marking M6.

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. Let us examine the 4 cases:

1. if there is a node with marking m1 = m then covered(L, γ) = true and the node L will be marked as processed
2. if there is a node with marking m1 < m then covered(L, γ) = true and the node L will be marked as processed
3. let us first consider the case that there is an ancestor of m such that m1 < m. In that case the whistle(L, γ) will blow (by Lemma 2) and return the first node on the path from the root such that m1 < m (m1 = m is not possible as case 1 hasn’t been applied). By Lemma 3 abstract(L, W, γ) we now know that abstract(L, W, γ) will give exactly m2

...
B. A KARP-MILLER TREE USING ECCE

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

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 PNI of [11].