Model Checking LTL using Constraint Programming*

Javier Esparza and Stephan Melzer

Institut für Informatik, Arcisstraße 21
Technische Universität München, D-80333 München, Germany
e-mail: {esparza,melzers}@informatik.tu-muenchen.de

Abstract. The model-checking problem for 1-safe Petri nets and linear-time temporal logic (LTL) consists of deciding, given a 1-safe Petri net and a formula of LTL, whether the Petri net satisfies the property encoded by the formula. This paper introduces a semidecision test for this problem. By a semidecision test we understand a procedure which may answer 'yes', in which case the Petri net satisfies the property, or 'don't know'. The test is based on a variant of the so called *automata-theoretic approach* to model-checking and on the notion of T-invariant. We analyse the computational complexity of the test, implement it using *2lp* – a constraint programming tool, and apply it to two case studies.
This paper is a (very) abbreviated version of [6].

1 Introduction

Linear-time temporal logic (LTL) is a well-known formalism for specifying properties of concurrent systems. The problem of deciding if a concurrent system satisfies a LTL formula is called the *model-checking problem* (of LTL). In [16] Vardi and Wolper introduced an *automata-theoretic approach* to this problem. The approach assumes that there exists a semantic mapping which associates to a concurrent system *sys* a *finite* (labelled) transition system A_{sys}. It asks the verifier to perform the following three tasks [9, 16]:

- Build a Büchi automaton $A_{\neg\phi}$ for the negation of the formula ϕ to be checked. $A_{\neg\phi}$ accepts exactly all infinite sequences that violate the formula ϕ.
- Construct a Büchi automaton A_p, called the *product* of A_{sys} and $A_{\neg\phi}$. A_p accepts all the infinite computations of A_{sys} that are accepted by $A_{\neg\phi}$, i.e., all infinite computations of A_{sys} that violate ϕ.
- Check whether the product automaton A_p is empty, i.e., whether it accepts no infinite sequences. A_{sys} satisfies ϕ iff A_p is empty.

The main problem of this approach is the well-known state-explosion phenomenon: the size of the transition system A_{sys} can grow exponentially in the size of *sys*. Several suggestions have been made to solve or at least palliate this problem: the transition system A_{sys} can be replaced by a trace automaton [9],

* This work is supported by the Sonderforschungsbereich SFB-342 A3.

and the size of A_{sys} can be reduced by means of different techniques like stubborn sets [14], sleep sets [9], or others.

In this paper we introduce still another technique to avoid the state-explosion, which can be applied when the system is modelled as a 1-safe Petri net. The technique is a *semidecision test*, that is, a procedure which may answer 'yes', in which case the property to be checked holds, or 'don't know'. A semidecision test has interest only if for relevant case studies it answers 'yes' and performs faster than exact methods. We provide evidence in this direction in the form of a complexity analysis and two case studies.

For systems modelled as Petri nets the transition system A_{sys} is just the well-known reachability graph. An straightforward application of the automata-theoretic approach would proceed by (1) building the reachability graph, and by (2) constructing the product automaton; it would obviously suffer from the state explosion problem. The first (minor) contribution of this paper is to show that step (2) can be performed before step (1). More specifically, we describe several ways of constructing a 'product Büchi net' N_p from a Petri net N_{sys} and a Büchi automaton $A_{\neg\phi}$. Using this construction it is immediate to reduce the model-checking problem to a certain 'net emptiness' problem, very similar to the emptiness problem of Büchi automata. We select the construction of the product Büchi net most suitable for our semidecision test. The test is based on the notion of T-invariant, and can be seen as a generalization of the ad-hoc proof method introduced and applied in [7]. We show that the test can be implemented in the framework of constraint programming [12] using the constraint programming tool 2lp [13]. Finally, we apply the test to a leader election and to a snapshot algorithm.

The paper is organised as follows. Section 2 describes the main components of the automata-theoretic approach to model-checking, tailored for the case in which the system is modelled by a Petri net. Section 3 shows how to construct the product Büchi nets. Section 4 introduces the test for net emptiness. Section 5 contains the implementation in 2lp. Section 6 is devoted to the case studies.

The paper is an abbreviated version of [6]. The reader can find there the proofs of the results, a detailed description of the case studies, and additional results.

2 The automata-theoretic approach to model-checking

2.1 Transition systems

A *labelled transition system* is a fourtuple (Act, Q, Δ, q_0), where Act is an alphabet of *actions*, Q is a set of *states*, $\Delta \subseteq Q \times Act \times Q$ is a set of *transitions*, and $q_0 \in Q$ is the *initial state*.

A *full run* of a labelled transition system is an infinite sequence $q_0 a_0 q_1 a_1 q_2 \ldots$ such that $(q_i, a_i, q_{i+1}) \in \Delta$ for every $i \geq 0$. We also denote a full run by $q_0 \xrightarrow{a_0} q_1 \xrightarrow{a_1} q_2 \ldots$.

When labelled transition systems are used as semantics of some process algebra only the labels of the transitions carry useful information; the intermediate states are usually irrelevant. We speak in this case of an *action-based* semantics. In action-based semantics the following definition is useful: An infinite sequence $a_0 a_1 a_2 \ldots$ of actions of $\mathcal{T}$ is an *action run* if there exists a full run $q_0 \xrightarrow{a_0} q_1 \xrightarrow{a_1} q_2 \xrightarrow{a_2} \ldots$. The *action language* $L_a(\mathcal{T})$ of $\mathcal{T}$ is the set of all action runs.

When labelled transition systems are used as semantics of languages with variables, the information about the actual values of the variables is encoded into the states; the labels of the transitions are usually irrelevant. We speak in this case of a *state-based* semantics. In state-based semantics the following definition is useful: An infinite sequence $q_0 q_1 q_2 \ldots$ of states of $\mathcal{T}$ is a *state run* if there exists a full run $q_0 \xrightarrow{a_0} q_1 \xrightarrow{a_1} q_2 \ldots$. The *state language* $L_s(\mathcal{T})$ of $\mathcal{T}$ is the set of all state runs.

For state-based semantics it is convenient to use (unlabelled) transition systems instead of carrying a useless action set Act around. An (unlabelled) *transition system* is a tuple (Q, Δ, q_0), where $\Delta \subseteq Q \times Q$. It can be seen as a particular case of labelled transition system in which all transitions carry the same label.

In the paper we use $L(\mathcal{T})$ to denote any of $L_a(\mathcal{T})$ or $L_s(\mathcal{T})$.

2.2 Linear-time Temporal Logic

Let Σ be a finite alphabet, and let Π be a set of propositions over Σ, i.e., a set of mappings with Σ as domain and the set $\{true, false\}$ as range. The set of formulae of *linear-time propositional temporal logic* (LTL) over the set Π is inductively defined as follows:

- if $\phi \in \Pi$ then ϕ is a formula
- if ϕ and ψ are formulae then so are $\phi \wedge \psi$, $\neg\phi$, $X\phi$ and $\phi \mathbin{\mathsf{U}} \psi$.

We make use of the abbreviations $\phi \vee \psi = \neg(\neg\phi \wedge \neg\psi)$, $\phi \mathbin{\mathsf{V}} \psi = \neg(\neg\phi \mathbin{\mathsf{U}} \neg\psi)$, $\diamond\phi = \mathbf{true} \mathbin{\mathsf{U}} \phi$ and $\square\phi = \neg \diamond \neg\phi$. An interpretation of an LTL-formula is an infinite word $\xi \in \Sigma^\omega$. In order to formally define the satisfaction relation $\models$ of LTL, let $\xi(0)$ denote the first element of ξ, and let $\xi^{(i)}(x) = \xi(x+i)$ denote the suffix of ξ starting at position i. We have:

- $\xi \models \pi$ for $\pi \in \Pi$ if $\pi(\xi(0)) = true$.
- $\xi \models \neg\phi$ if not $\xi \models \phi$.
- $\xi \models \phi \wedge \psi$ if $\xi \models \phi$ and $\xi \models \psi$.
- $\xi \models X\phi$ if $\xi^{(1)} \models \phi$.
- $\xi \models \phi \mathbin{\mathsf{U}} \psi$ if $\exists i \in \mathbb{N} \ : \ \xi^{(i)} \models \psi$ and $\forall j \leq i \ : \ \xi^{(j)} \models \phi$.

The *language* $L(\phi)$ of a formula ϕ over Π is the set of all words of Σ^ω that satisfy ϕ.

2.3 LTL on transition systems

We wish to use LTL to describe properties of both the action-based and the state-based semantics of a labelled transition system $\mathcal{T} = (Act, Q, \Delta, q_0)$. In the case of action-based semantics, we take $\Sigma = Act$. Π is therefore a set of propositions on the set of *actions*, and the language $L(\phi)$ of a formula ϕ is a set of action runs. We say that $\mathcal{T}$ satisfies ϕ if $L_a(\mathcal{T}) \subseteq L(\phi)$, i.e., if every action run of $\mathcal{T}$ satisfies ϕ. In state-based semantics, we take $\Sigma = Q$, and so Π is a set of propositions on the set of *states*. Analogously, we say that $\mathcal{T}$ *satisfies* ϕ if $L_s(\mathcal{T}) \subseteq L(\phi)$.

2.4 Büchi Automata

Let ϕ be a formula of LTL over a set of propositions Π. A *labelled Büchi automaton* over Π is a tuple $A = (2^\Pi, Q, \Delta, q_0, F)$, where Q is a finite set of *states*, $\Delta \subseteq Q \times 2^\Pi \times Q$ is the *transition relation*, $q_0 \in Q$ is the *initial state*, and $F \subseteq Q$ is the set of *accepting states*. An *accepting run* of A is an infinite sequence $\sigma = q_0 \Pi_0 q_1 \Pi_1 q_2 \ldots$ such that $(q_i, \Pi_i, q_{i+1}) \in \Delta$ for every $i \geq 0$, and some state of F appears infinitely often in σ. A *accepts* an infinite word $a_0 a_1 a_2 \ldots \in \Sigma^\omega$ if there exists an accepting run $q_0 \Pi_0 q_1 \Pi_1 q_2 \ldots$ such that a_i satisfies every predicate of Π_i, for every $i \geq 0$.

We define the *language* $L(A)$ of a labelled Büchi automaton A as the set of infinite words accepted by A.

We have the following important result:

Theorem 1 [15]. *Let ϕ be a formula of LTL. There exists a Büchi automaton A such that $L(\phi) = L(A)$*

In the sequel we use A_ϕ to denote a Büchi automaton satisfying $L(\phi) = L(A_\phi)$, which we assume has been constructed using some algorithm, for instance the one described in [8].

We also use *unlabelled Büchi automata*, which are tuples $A = (Q, \Delta, q_0, F)$, where $\Delta \subseteq Q \times Q$. They can be seen as a special case of labelled Büchi automata in which all transitions are labelled by the empty set of propositions.

The *nonemptiness problem* for a labelled or unlabelled Büchi automaton A consists of deciding whether $L(A)$ is nonempty. The problem is NLOGSPACE-complete [15].

2.5 Product automata

Let $\mathcal{T}_{sys}$ be a finite labelled transition system, and let ϕ be a formula over the actions or the states of $\mathcal{T}_{sys}$. The automata-based procedure to check if $\mathcal{T}_{sys}$ satisfies ϕ consists of the following steps:

- Build a labelled Büchi automaton $A_{\neg\phi}$ which accepts $L(\neg\phi)$.
- Build an unlabelled Büchi automaton A_p, called the *product* of $\mathcal{T}_{sys}$ and $A_{\neg\phi}$, which is empty iff $L(\mathcal{T}_{sys}) \cap L(\neg\phi) = \emptyset$.
- Check whether $L(A_p)$ is nonempty.

Clearly, $L(A_p)$ is empty iff $L(\mathcal{T}_{sys}) \cap L(\neg\phi) = \emptyset$ iff $L(\mathcal{T}_{sys}) \subseteq L(\phi)$ iff $\mathcal{T}_{sys}$ satisfies ϕ.

The following two subsections show how to construct A_p for action-based and state-based semantics.

Action-based semantics Let $\mathcal{T}_{sys} = (Act_{sys}, Q_{sys}, \Delta_{sys}, q_{osys})$ be a labelled transition system, and let $A_{\neg\phi} = (2^{\Pi}, Q_{\neg\phi}, \Delta_{\neg\phi}, q_{o\neg\phi}, F_{\neg\phi})$ be the labelled Büchi automaton corresponding to the negation of ϕ, where Π is a set of propositions on Act_{sys}. The product automaton of $\mathcal{T}_{sys}$ and $A_{\neg\phi}$ is the unlabelled Büchi automaton $A_p = (Q, \Delta, q_0, F)$ given by

- $Q = Q_{sys} \times Q_{\neg\phi}$,
- Δ is the smallest set such that if $(q_1, a, q_2) \in \Delta_{sys}$, $(r_1, \{\pi_1, \ldots, \pi_n\}, r_2) \in \Delta_{\neg\phi}$, and a satisfies π_i for every $1 \leq i \leq n$, then $((q_1, r_1), (q_2, r_2)) \in \Delta$.
- $q_0 = (q_{0sys}, q_{0\neg\phi})$,
- $F = Q_{sys} \times F_{\neg\phi}$.

It follows immediately from this definition that A_p is empty if and only if the set $L_a(\mathcal{T}_{sys}) \cap L(A_{\neg\phi}) = L_a(\mathcal{T}_{sys}) \cap L(\neg\phi)$ is also empty.

State-based semantics Let $\mathcal{T}_{sys} = (Q_{sys}, \Delta_{sys}, q_{0sys})$ be an unlabelled transition system, and let $A_{\neg\phi} = (2^{\Pi}, Q_{\neg\phi}, \Delta_{\neg\phi}, q_{o\neg\phi}, F_{\neg\phi})$ be the labelled Büchi automaton corresponding to the negation of ϕ, where Π is a set of propositions on Q_{sys}. The product automaton of $\mathcal{T}_{sys}$ and $A_{\neg\phi}$ is the unlabelled Büchi automaton $A_p = (Q, \Delta, q_0, F)$ given by

- $Q = Q_{sys} \times Q_{\neg\phi}$,
- Δ is the smallest set such that if $(q_1, q_2) \in \Delta_{sys}$, $(r_1, \{\pi_1, \ldots, \pi_n\}, r_2) \in \Delta_{\neg\phi}$ and q_1 satisfies π_i for every $1 \leq i \leq n$, then $((q_1, r_1), (q_2, r_2)) \in \Delta$,
- $q_0 = (q_{0sys}, q_{0\neg\phi})$,
- $F = Q_{sys} \times F_{\neg\phi}$.

The only difference with the former definition is the fact that the propositions π_i are now evaluated on the state q_1, and not on the action a.

Again, it follows immediately from this definition that A_p is empty if and only if the set $L_s(\mathcal{T}_{sys}) \cap L(A_{\neg\phi})$ is also empty.

3 Lifting the automata-theoretic model-checking method to Petri nets

3.1 Multiset Notation

A *multiset* over a set X is a mapping $\mu : X to \mathbb{N}$. The operations union, intersection, sum, and difference on multisets are defined in the usual way (see for instance [1]). The set of multisets over X is denoted by $\mathcal{M}(X)$.

A *labelled Petri net* is a tuple $N = (Act, P, T, M_0)$ where Act is a set of *actions*, P is a finite set of *places*, $T \subseteq (\mathcal{M}(P) \times Act \times \mathcal{M}(P))$ is a set of *transitions*, and $M_0 \in \mathcal{M}(P)$ is a *marking*. For a transition $t = (P, Q)$ we sometimes call P (resp. Q) the *preset* (resp. *postset*) and write ${}^\bullet t$ (resp. $t^\bullet$). Multisets of places are called *markings*, and M_0 is called the *initial marking* of N.

Notions like enabled transition, firing, reachable marking, 1-safe Petri net (also called safe or 1-bounded Petri net), incidence matrix, T-invariant and P-component (also called S-component) are defined as usual (see for instance [5]).

$M \xrightarrow{t} M'$ denotes that transition t occurs at marking M yielding M'. A finite or infinite sequence $M_0 \xrightarrow{t_0} M_1 \xrightarrow{t_1} M_2 \ldots$ is called an *occurrence sequence*. $M \xrightarrow{a} M'$ for $a \in \Sigma$ denotes that there exists a transition $t = (P_1, a, P_2)$ such that $M \xrightarrow{t} M'$.

A *full run* of a Petri net is an infinite sequence $M_0 a_0 M_1 a_1 M_2 a_2 \ldots$ such that $M_i \xrightarrow{a_i} M_{i+1}$ for every $i \geq 0$. We also denote a full run by $M_0 \xrightarrow{a_0} M_1 \xrightarrow{a_1} M_2 \ldots$. Notice that for every full run there exists an underlying occurrence sequence.

An infinite sequence $a_0 a_1 a_2 \ldots$ of actions is an *action run* if there exists a full run $M_0 \xrightarrow{a_0} M_1 \xrightarrow{a_1} M_2 \ldots$. The *action language* $L_a(N)$ of N is the set of all action runs. An infinite sequence $M_0 M_1 M_2 \ldots$ of markings is a *state run* if there exists a full run $M_0 \xrightarrow{a_0} M_1 \xrightarrow{a_1} M_2 \ldots$. The *state language* $L_s(N)$ of N is the set of all state runs.

As usual, unlabelled Petri nets are obtained from labelled ones by dropping the labelling of transitions. So an unlabelled Petri net is a tuple (P, T, M_0) where $T \subseteq \mathcal{M}(P) \times \mathcal{M}(P)$.

If we are only interested in the structure of a Petri net, then we omit M_0 and call (P, T) just a net.

3.2 LTL on 1-safe Petri nets

We define when a 1-safe Petri net satisfies a formula of LTL. In action-based semantics Π is a set of propositions on the set of actions of the Petri net. As for transition systems, we say that a net N satisfies a formula ϕ if $L_a(N) \subseteq L(\phi)$, i.e., if every action-based run of N satisfies ϕ.

The state-based case is more interesting. For transition systems, we let Π be a set of propositions on the set of states. Since the states of a Petri net are its reachable markings, for Petri nets we should take Π as an arbitrary set of propositions on the set of *markings*. However, we restrict ourselves to propositions π_p, where p is a place of the net, with the following interpretation: a marking M satisfies π_p iff it marks the place p. We say that a net N satisfies a formula ϕ if $L_s(N) \subseteq L(\phi)$.

It is easy to see that this restriction has no important consequences: the two logics we obtain (one with arbitrary propositions over markings, the other with the restricted set), have the same expressive power for 1-safe Petri nets [6].

3.3 Büchi Nets

The product of a Büchi automaton and a 1-safe Petri net is going to be a Büchi net, the net counterpart of the unlabelled product Büchi automaton defined in Section 2.5.

A *Büchi net* is a tuple $N = (P, T, M_0, F)$, where (P, T, M_0) is an unlabelled Petri net and F is a subset of P. An *accepting run* of N is a state run $M_0 M_1 M_2 \ldots$ such that some place of F appears in infinitely many markings M_i. N is *nonempty* if it has an accepting run.

The *nonemptiness problem* for a Büchi net $N = (P, T, M_0, F)$ is the problem of deciding if N is nonempty. It is easy to show that the problem is PSPACE-complete [6].

3.4 Product nets in action-based semantics

It is easy to lift the definition of the product automaton to the Petri net case.

Let $N_{sys} = (Act_{sys}, P_{sys}, T_{sys}, M_{0sys})$ be a 1-safe Petri net, and let $A_{\neg\phi} = (2^{\Pi}, Q_{\neg\phi}, \Delta_{\neg\phi}, q_{o\neg\phi}, F_{\neg\phi})$ be the Büchi automaton corresponding to the negation of ϕ, where Π is a set of propositions on Act_{sys}.

Definition 2. The product Büchi net $N_p = (P, T, M_0, F)$ of N_{sys} and $A_{\neg\phi}$ is given by

- $P = P_{sys} \cup Q_{\neg\phi}$,
- T is the smallest set satisfying: if $(P_1, a, P_2) \in T_{sys}$ and $(q_1, \{\pi_1, \ldots, \pi_n\}, q_2) \in \Delta_{\neg\phi}$, and $\pi_i(a)$ holds for every $1 \leq i \leq n$, then $(P_1 + \{q_1\}, P_2 + \{q_2\})) \in T$,
- $M_0 = M_{0sys} + \{q_{0\neg\phi}\}$,
- $F = F_{\neg\phi}$.

The following theorem is easy to prove:

Theorem 3 [6]. *Let N_{sys} be a 1-safe Petri net, and let $A_{\neg\phi}$ be the Büchi automaton corresponding to the negation of a property ϕ. Let N_p be as in Definition 2. N_p is 1-safe and N_{sys} satisfies ϕ iff N_p is empty.*

This same result holds for the other definitions of product we are going to present in the rest of this section (Definitions 2, 4, 5 and 9), and so the corresponding theorems are omitted. The theorems and their proofs can be found in [6].

3.5 Product nets in state-based semantics

We fix an unlabelled 1-safe Petri net $N_{sys} = (P_{sys}, T_{sys}, M_{0sys})$. We assume that the set Π of propositions on the markings of N_{sys} used to construct formulae of LTL contains only predicates π_p which hold iff the place p is marked. Clearly, we can (and will) identify the proposition π_p and the place p. With this identification, the Büchi automaton $A_{\neg\phi}$ for the negation of a formula ϕ has the form $A_{\neg\phi} = (2^{P_{sys}}, Q_{\neg\phi}, \Delta_{\neg\phi}, q_{0\neg\phi}, F_{\neg\phi})$.

Our goal is to construct a product Büchi net satisfying the following property: the product net can move from a marking (M_1, q_1) to (M_2, q_2) iff:

(1) N_{sys} can move from M_1 to M_2,

(2) there exists $(q_1, R, q_2) \in \Delta_{\neg\phi}$, and
(3) M_1 marks every place of R.

We show two different constructions. This first one is similar to that shown in section 2.5 for transition systems. The key idea is the following: if (P_1, P_2) is a transition of the Petri net and (q_1, R, q_2) is a transition of the Büchi automaton, then we add the following transition to the product:

$$(\ P_1 + (R - P_1) + \{q_1\} \ , \ P_2 + (R - P_1) + \{q_2\} \ )$$

It is immediate to see that this solution satisfies conditions (1) to (3) above. The product automaton can then be defined in the following way:

Definition 4. The product Büchi net $N_p = (P, T, M_0, F)$ of N_{sys} and $A_{\neg\phi}$ is given by

- $P = P_{sys} \cup Q_{\neg\phi}$,
- T is the smallest set satisfying: if $(P_1, P_2) \in T_{sys}$ and $(q_1, R, q_2) \in \Delta_{\neg\phi}$, then $(P_1 + (R - P_1) + \{q_1\}, P_2 + (R - P_1) + \{q_2\}) \in T$,
- $M_0 = M_{0sys} + \{q_{0\neg\phi}\}$,
- $F = F_{\neg\phi}$.

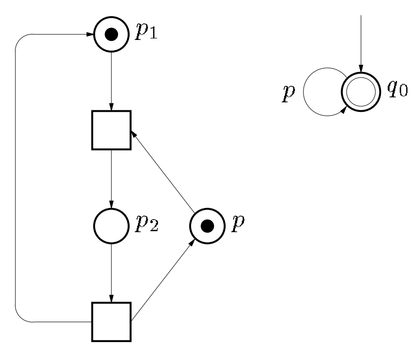

Fig. 1. A Petri net N_{sys} (lhs.) and a Büchi automaton $A_{\neg\phi}$ (rhs.).

Figure 2 illustrates this definition.

Loosely speaking, in the second construction the automaton and the Petri net alternate their moves: the automaton tests if the marking M_1 marks every place of R. If this is the case, then it moves from q_1 to q_2, and transfers controls to the net, who makes its move, and transfers control back to the automaton. The alternation can be implemented by means of two scheduling places SC_1, SC_2. A token on SC_1 (SC_2) means that the automaton (the net) has to move next.

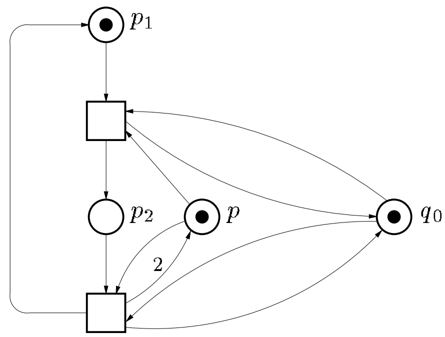

Fig. 2. The product net N_p of N_{sys} and $A_{\neg\phi}$ of Figure 1 w.r.t. Definition 4.

Definition 5. The product Büchi net $N_p = (P, T, M_0, F)$ of N_{sys} and $A_{\neg\phi}$ is given by

- $P = P_{sys} \cup Q_{\neg\phi} \cup \{SC_1, SC_2\}$,
- T is the smallest set satisfying: if $(P_1, P_2) \in T_{sys}$ then $(P_1 + \{SC_2\}, P_2 + \{SC_1\}) \in T$, and if $(q_1, R, q_2) \in \Delta_{\neg\phi}$ then $(\{q_1, SC_1\} + R, \{q_2, SC_2\} + R) \in T$;
- $M_0 = M_{0sys} + \{q_{o\neg\phi}, SC_1\}$,
- $F = F_{\neg\phi}$.

See Figure 3 for an example.

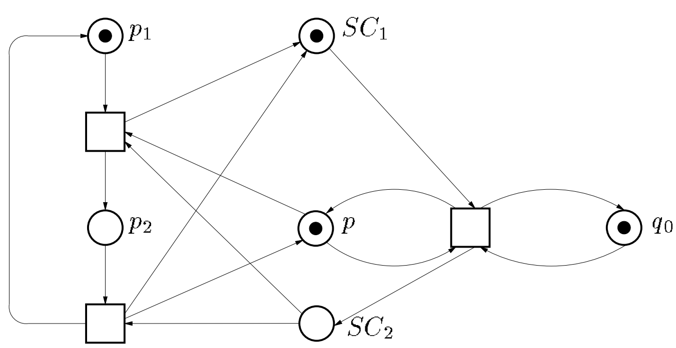

Fig. 3. The product net N_p of N_{sys} and $A_{\neg\phi}$ of Figure 1 w.r.t. Definition 5.

This second construction, contrary to the first, remains very small: its size is essentially the sum of the sizes of N_{sys} and $A_{\neg\phi}$. Unfortunately, as shown in the next section, this second construction faces other problems. We shall actually combine the two constructions in order to obtain good results.

4 Testing emptiness of Büchi nets using T-invariants

In Section 3 we have reduced the model-checking problem to the emptiness problem of Büchi nets. We now develop a semidecision test for this latter problem which avoids the construction of the reachability graph. The theory underlying the method is well-known; our contribution is a set of refinements and techniques for its application.

We have developed this test in order to verify parallel programs modelled in the language $B(PN)^2$ [2], which are automatically translated into 1-safe Petri nets by the PEP tool [10]. The fact that a variable x has a value v is modelled by putting a token on a place x_v. Therefore, assertions like "the variable x takes the value 1 infinitely often" are best formalised using state-based semantics. From now on we concentrate on this semantics, but the technique is also applicable (even more easily) to the action-based case.

The test is based on the notion of T-invariant. Recall that a T-invariant of a net is a mapping $\mathcal{J}$ that assigns to each transition t a rational number $\mathcal{J}(t)$ and satisfies the following property for every place p:

$$\sum_{t \in {}^\bullet p} \mathcal{J}(t) = \sum_{t \in p^\bullet} \mathcal{J}(t)$$

T-invariants have the following fundamental property. Let M and M' be markings of a net N, and let σ be a sequence of transitions such that $M \xrightarrow{\sigma} M'$. We have $M = M'$ iff the mapping which associates to each transition t the number of times that it appears in σ is a T-invariant of N.

A T-invariant $\mathcal{J}$ of a Büchi net N is *realisable* if there exists a reachable marking M and a nonempty sequence of transitions σ such that $M \xrightarrow{\sigma} M$ and every transition t occurs exactly $\mathcal{J}(t)$ times in σ. The sequence $M \xrightarrow{\sigma} M$ is called a *realisation* of $\mathcal{J}$. Realisable T-invariants are always *semi-positive*, i.e., its components have to be nonnegative, and at least one of them must be different from 0. A T-invariant $\mathcal{J}$ is *final* if $\mathcal{J}(t) > 0$ for some transition t in the postset of a final place of N. The following result is easy to prove:

Proposition 6 [6]. *A Büchi net is nonempty iff it has a final realisable T-invariant.*

As an immediate consequence of this proposition, if a Büchi net has no final semi-positive T-invariants, realisable or not, then it is empty. This sufficient condition for emptiness leads to a simple semidecision test, since the absence of semi-positive T-invariants can be checked by solving a system of linear (in)equations of the form

$$\begin{array}{l} \mathbf{N} \cdot X = 0 \\ X \geq 0 \\ \sum_{t \in F^\bullet} X(t) > 1 \end{array}$$

where $\mathbf{N}$ is the incidence matrix of the Büchi net, and F is the set of final places.

The practical interest of a semidecision test is directly proportional to its *quality* (i.e., how often it is successful, or, in our case, how often does it prove emptiness) and inversely proportional to its computational complexity. It is well-known that systems of linear (in)equations can be solved very efficiently using the simplex algorithm, and in guaranteed polynomial time by other techniques. So the test above is very efficient. Unfortunately, its quality is very low. In nearly all examples of interest the test fails to provide an answer even if the language of the net is empty. So we refine Definition 4 in order to improve the quality of the test. In subsection 4.1 we observe that some of the transitions of N_p can never occur. Since these transitions never appear in any infinite occurrence sequence of N_p, they can be removed without affecting the result stating that N_{sys} satisfies ϕ iff N_p is empty. Clearly, after removing this transitions the resulting net has exactly the same realisable T-invariants, but *less* semipositive T-invariants, which improves the quality of the test.

Unfortunately, with the improved definition of product the number of transitions of N_p can still be unacceptably large, similarly to what happened in the action-based case. In Section 4.2 we show that this problem can be palliated by combining the improved Definition 4 with Definition 5.

4.1 Removing dead transitions

Let N_p be a product net obtained according to Definition 4, and let $t = (P_1 + (R - P_1) + \{q_1\}, P_2 + (R - P_1) + \{q_2\})$ be a transition such that there exists a place $p \in (P_2 - P_1) \cap R$. It is shown in [6] that t can never occur in N_p.

This is how far we can go if we have no other information about N_{sys}. However, we often know that N_{sys} has a certain set of P-components which contain exactly one token at the initial marking. Recall that a *P-component* is a connected subnet in which every transition has exactly one input and one output place, and which is connected to other nodes of the net only through transitions[2]. The number of tokens of a P-component remains constant under the occurrence of transitions.

Information about the P-components of the net is very often available in practice. Systems modelled by 1-safe nets are usually composed by several sequential systems that communicate via message passing, rendezvous, or shared variables. In all cases, the models of these components are P-components of the global model.

Let $N_i = (P_i, T_i)$ be a P-component carrying exactly one token at the initial marking, and let

$$t = (P_1 + (R - P_1) + \{q_1\}, P_2 + (R - P_1) + \{q_2\})$$

be a transition such that $|(P_1 + (R - P_1)) \cap P_i| > 1$. It is shown in [6] that t can never occur in N_p.

[2] Sometimes P-components are also required to be strongly connected subnets, but that is not necessary in our case.

The transition with the double weighted arc in Figure 2 can be removed using this criterion.

We introduce the following definition:

Definition 7. $(P_1, P_2) \in T_{sys}$ and $(q_1, R, q_2) \in \Delta_{\neg\phi}$ are *compatible* if the two following properties hold:

- $(P_2 \cap R) \subseteq (P_1 \cap R)$, and
- for all $1 \leq i \leq k$: if $(P_1 \cap P_i) \neq \emptyset$ and $(P_i \cap R) \neq \emptyset$, then $(P_1 \cap P_i) = (P_i \cap R)$.

If (P_1, P_2) and (q_1, R, q_2) are compatible, then we also say that (q_1, R, q_2) is compatible with (P_1, P_2), or that (P_1, P_2) is compatible with (q_1, R, q_2).

Now, in Definition 4 we can substitute the description of the set T by the following one:

- T is the smallest set satisfying: if $(P_1, P_2) \in T_{sys}$ and $(q_1, R, q_2) \in \Delta_{\neg\phi}$ are compatible, then $(P_1 + (R - P_1) + \{q_1\}, P_2 + (R - P_1) + \{q_2\}) \in T$.

4.2 Combining Definition 4 and Definition 5

Let (P_1, P_2) be a transition that is compatible with every transition of $A_{\neg\phi}$. With respect to (P_1, P_2), the new definition of product coincides with the old one: the same set of transitions of the product is generated. However, n of these transitions generate $n \cdot |T_{\neg\phi}|$ transitions in the product net, which can be unacceptable if n is large.

The solution to this problem is to use the product discipline of Definition 5 for these transitions, and reserve the discipline of Definition 4 for those which can improve the quality of the test. In order to implement this idea we need the following definition:

Definition 8. A transition (P_1, P_2) of N_{sys} is *compatible* with $A_{\neg\phi}$ if it is compatible with every transition of $\Delta_{\neg\phi}$.

Definition 9. The product Büchi net $N_p = (P, T, M_0, F)$ of N_{sys} and $A_{\neg\phi}$ is given by

- $P = P_{sys} \cup Q_{\neg\phi} \cup \{SC_1, SC_2\}$,
- T is the smallest set satisfying:
  (1) if $(q_1, R, q_2) \in \Delta_{\neg\phi}$, then $(R \cup \{q_1, SC_1\}, R \cup \{q_2, SC_2\}) \in T$,
  (2) if $(P_1, P_2) \in T_{sys}$ is compatible with $A_{\neg\phi}$, then $(P_1 + \{SC_2\}, P_2 + \{SC_1\}) \in T$,
  (3) if $(P_1, P_2) \in T_{sys}$ is not compatible with $A_{\neg\phi}$, then $(P_1 + (R - P_1) + \{q_1, SC_1\}, P_2 + (R - P_1) + \{q_2, SC_1\}) \in T$ for every $(q_1, R, q_2) \in \Delta_{\neg\phi}$ compatible with (P_1, P_2).
- $M_0 = M_{0sys} + \{q_{o\neg\phi}, SC_1\}$,
- $F = F_{\neg\phi}$.

4.3 An improved test

In this section we introduce the notion of T*-invariant, and use it to develop a new emptiness test. The quality is improved at the price of more computational complexity: the new test is NP-complete. The quality will be now good enough for verifying interesting liveness properties of real systems.

One of the main reasons why the test of the previous section has a low quality is the fact that the Büchi nets we wish to analyse usually contain self-loops, i.e., they contain places that are both input and output places of transitions. The presence of self-loops may lead to the typical situation shown in Figure 4. The

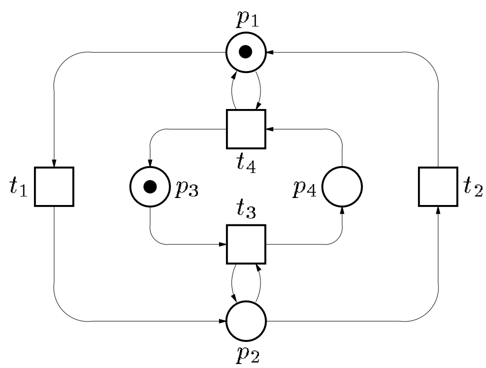

Fig.4. Net with selfloops.

vector $\mathcal{J} = (0,0,1,1)^t$ is a T-invariant, but not a realisable T-invariant. To prove it, observe that the subnet N' generated by the places $\{p_1, p_2\}$ and the transitions $\{t_1, \ldots t_4\}$ is a P-component (see Figure 6), and so $M(p_1) + M(p_2) = 1$ holds for every reachable marking M. Now, assume that $\mathcal{J}$ is realisable. Then it has a realisation $M \xrightarrow{\sigma} M$. Since $\mathcal{J} = (0,0,1,1)^t$, σ only contains occurrences of t_3 and t_4. It is easy to see that the projection $M' \xrightarrow{\sigma'} M'$ of $M \xrightarrow{\sigma} M$ onto the places and transitions of N' is an occurrence sequence of N'. But this leads to a contradiction: since t_3 needs a token on p_2 to occur, and t_4 needs a token on p_1, t_3 can never occur immediately after t_4; the transition t_1 must occur inbetween. Similarly, t_4 can never occur immediately after t_3; the transition t_2 must occur inbetween. More generally, the subnet of N' generated by transitions t_1 and t_2 together with their input and output places (shown in Figure 5) is *not strongly connected*, and therefore no sequence containing only t_3 and t_4 can be an occurrence sequence of N'. This shows that $\mathcal{J}$ is not realisable. In this proof we have used again information about the P-components of the net, namely the fact that N' is a P-component which carries initially one single token. This leads to the following definition:

Definition 10. Let $N = (P, T)$ be a net and let $N_i = (P_i, T_i)$, $1 \leq i \leq n$ be a

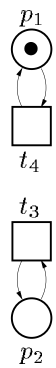

Fig. 5. The subnet N'.

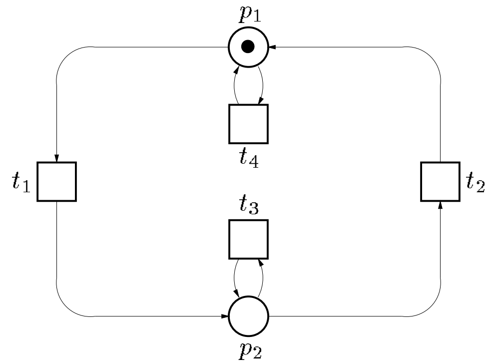

Fig. 6. The P-component with places $\{p_1, p_2\}$.

set of P-components of N. We call a T-vector $\mathcal{J}$ a *T^*-invariant with respect to* $N_1, \ldots, N_k$ if

- $\mathcal{J}$ is a semi-positive T-invariant, and
- for every $1 \leq i \leq n$, the subnet of N_i generated by the transitions of T_i that appear in $\mathcal{J}$, together with their input and output places, is strongly connected.

The T-invariant $(0, 0, 1, 1)^t$ above is *not* a T*-invariant with respect to N', because the subnet of Figure 5 is not strongly connected. It is easy to see that realisable T-invariants are T*-invariants with respect to any set of P-components carrying one token [6]. This implies:

Theorem 11 [6]. *Let N be a Büchi net and let N_i, $1 \leq i \leq n$ be a set of P-components of N carrying one token at the initial marking. If N has no final T^*-invariants with respect to $N_1, \ldots, N_k$, then it is empty.*

We call the problem of deciding the existence of a T*-invariant for a given net and a given set of P-components the *T*- invariant problem.* We have:

Theorem 12 [6]. *The T*-invariant problem is NP-complete.*

5 An Implementation of the T*-Invariant Test Using Constraint Programming

A system of linear inequations can be seen as a *conjunction* of linear constraints i.e., the feasible region of the system (its set of solutions) is the set of vectors that satisfy all the constraints.

We can thus interpret linear programming as a primitive constraint programming language, in which the only available operator to combine constraints is

AND. Simplex, or any other algorithm for linear programming, can be seen as an inference engine for this programming language.

While the emptiness test based on traditional T-invariants can be implemented in linear programming, this is no longer true for the T*-invariant problem: the AND construct is not powerful enough.

Fortunately, in the last years there have been a number of efforts to develop programming environments for linear and integer programming that goes well beyond the AND construct. One of these environments is 2lp [13]. Citing from [13]: "2lp is a constraint logic programming language [12] with C-like syntax which can be used to make linear and integer programming part of programming in the contemporary sense of the word".

An adequate introduction to 2lp is out of the scope of this paper; we refer the interested reader to [13]. For our purposes, it suffices to know that the semantics of a 2lp program is a (not necessarily linear) *constraint* on the space of its variables, or, equivalently, a *feasible region* (the tuples of values of the variables that satisfy the constraint). 2lp contains different operators to produce complex constraints out of simpler ones. We introduce two of these operators in the following example:

```
3x − 2y = 1;
either {x + y ≤ 3}
    or {2x − y ≥ 3}
```

The operator ";" corresponds to the AND of linear programming. That is, the feasible region of the program above is the *intersection* of the feasible regions of $3x - 2y = 1$ and the `either` ...`or` constraint. The feasible region of the `either` ...`or` constraint is the *union* of the feasible regions of the constraints $x + y \leq 3$ and $2x - y \geq 3$.

2lp also provides an operator to test the consistency of sets of constraints:

```
x ≤ y + 3;
y ≤ 3x − 5;
if not x = y then printf(''Inconsistent'')
else printf(''Consistent'')
```

The feasible region associated to this program is the feasible region of its first two constraints (i.e, the `not` operator does not change the feasible region). However, the `if not` ...`then` ...`else` instruction determines if the constraint $x = y$ is consistent with the first two, and answers accordingly.

We use these features to build a 2lp program that decides if a net contains a final T*-invariant with respect to a set of P-components. To lighten the notation, we consider only the case in which the set contains only one component. The general case is similar.

We start by "massaging" the condition in the definition of T*-invariants concerning strong connectedness. Fix a net $N = (P, T)$ and a P-component $N' = (P', T')$ of N, and let $U \subseteq T'$. Think of U as the intersection of T' and the set of transitions of a given T-invariant, of which we would like to determine if it is also a T*-invariant. Let N'_U be the subnet of N' generated by U and $P' \cap ({}^\bullet U \cup U^\bullet)$. We wish to know whether N'_U is strongly connected or not.

Define the relation $\leadsto_U \subseteq T' \times T'$ as follows: $t \leadsto_U t'$ if $t, t' \in U$, and there exists a place $p \in P'$ such that $t \in {}^\bullet p$ and $t' \in p^\bullet$. A set $V \subseteq U$ is *closed* under $\leadsto_U$ if $t \in V$ and $t \leadsto_U t'$ implies $t' \in V$. Notice that U is trivially closed under $\leadsto_U$.

We have the following lemma:

Lemma 13 [6]. *N'_U is strongly connected iff the only nonempty subset of U that is closed under $\leadsto_U$ is U itself.*

We now define several sets of constraints on the following variables: a vector $\mathtt{J} \in \mathbb{Q}^{|T|}$, and two boolean vectors $\mathtt{U}, \mathtt{V} \in \{0,1\}^{|T|}$, where we interpret the values of $\mathtt{U}$ and $\mathtt{V}$ as subsets of P'.

Each set of constraints is to be understood conjunctively, i.e., as if its elements were linked by AND, or by the semicolon of 2lp.

(1) $\mathtt{J}$ is a semipositive T-invariant. For each $p \in P$:

$$\sum_{t \in {}^\bullet p} \mathtt{J[t]} = \sum_{t \in p^\bullet} \mathtt{J[t]}$$

and for each $t \in T$:

$$J[t] \geq 0$$

(2) $\mathtt{J}$ is final.

$$\sum_{t \in F^\bullet} \mathtt{J[t]} > 0$$

(3) $\mathtt{U}$ is the intersection of T' and the support of $\mathtt{J}$. For each $t \notin T'$:

$$\mathtt{U_i[t]} = 0$$

and for each $t \in T'$:

$$\begin{aligned} \mathtt{either}\ \{\mathtt{J[t]} > \mathtt{0}; \mathtt{U[t]} = \mathtt{1}\} \\ \mathtt{or}\ \{\mathtt{J[t]} = \mathtt{0}; \mathtt{U[t]} = \mathtt{0}\} \end{aligned}$$

(4) $\mathtt{V}$ is a subset of $\mathtt{U}$. For each $t \in T$:

$$\begin{aligned} \mathtt{V}[t] \leq \mathtt{U}[t] \\ \mathtt{either}\ \mathtt{V}[t] = 1 \\ \mathtt{or}\ \mathtt{V}[t] = 0 \end{aligned}$$

(5) $\mathtt{V}$ is nonempty.

$$\sum_{t \in T} \mathtt{V}[t] > 0$$

(6) $\mathtt{V}$ is closed under $\leadsto_U$. For each $t, t' \in T'$ such that there exists $p \in P'$ satisfying $t \in {}^\bullet p$ and $t' \in p^\bullet$:

$$\mathtt{V}[t] + \mathtt{U}[t'] \leq 1 + \mathtt{V}[t']$$

(this constraint is the linear equivalent of $(t \in V \wedge t' \in U) \rightarrow t' \in V$)

(7) $\mathtt{V}$ contains less transitions than $\mathtt{U}$.

$$\sum_{t \in T} \mathtt{V}[t] < \sum_{t \in T} \mathtt{U}[t]$$

Now, define the 2lp program $\mathcal{LOG}_N$ as

```
(1); (2); (3);
not {(4); (5); (6); (7)}
```

The feasible region of (1) and (3) is the set of triples $(\mathcal{J}, U, V)$ where $\mathcal{J}$ is a final semipositive T-invariant and U is the intersection of T' and the support of J. The feasible region of (4) to (7) is the set of triples $(\mathcal{J}, U, V)$ where V is a proper and nonempty subset of U closed under $\leadsto_U$. According to the semantics of the `not` construct, $\mathcal{LOG}_N$ answers "No T*-invariants wrt. N'" iff the conjunction of the constraints (4) to (7) is inconsistent with the conjunction of (3) and (6). Therefore, $\mathcal{LOG}_N$ answers "No T*-invariants wrt. N'" iff for every final semipositive T-invariant the only nonempty subset of U closed under $\leadsto_U$ is U itself. This is the case iff N contains no final T*-invariants wrt. N'.

6 Applications

In this section we demonstrate the applicability of our verification method by means of two examples. We first consider a (variant of a) ring election algorithm designed by Chang and Roberts [4]. Then, we verify Bouge's snapshot algorithm [3]. The algorithms have been encoded in $B(PN)^2$ (Basic Petri Net Programming Notation) [2], an imperative language designed to have a simple Petri net semantics. The code can be found in [6]. $B(PN)^2$ are automatically compiled into 1-safe Petri nets by the PEP-tool.

A ring election algorithm Consider a distributed system which consists of N processes $P_0, \ldots, P_{N-1}$ connected via a token ring. The ring election algorithm of Chang and Roberts allows the processes to agree on a master process. In our implementation we use a boolean variable *success* to indicate that some master process is found during a single ring election. After resetting all processes *success* is set to **false**.

Verification and results The main liveness property of the specification of the ring election is that a master process is found infinitely often. The corresponding LTL-formula is $\Box \diamond (success = \mathbf{true})$. We have verified this property for $N = 1 \ldots 10$ (N is the number of processes and fifo queues). Table 6 summaries the sizes of the original Petri net N_{sys} and the product Büchi net N_p for some representative values of N, together with the time needed to verify the absence of T*-invariants compared to the time SPIN [11] needed to verify the property. This example is particularly favourable to our technique due to the fact that there exist no semipositive T-invariants containing transitions in the pre- or the postset of the accepting places of the underlying Büchi automaton. It must also be said that the table does not include the time needed to construct the Petri net from the BPN^2 program. This time was very large (about half an hour for $N = 10$), but this is due to the fact that the implementation of the PEP-compiler from BPN^2 into Petri nets has not been optimized yet.

N	N_{sys}		N_p		time (sec.)	
	$\|P\|$	$\|T\|$	$\|P\|$	$\|T\|$	2lp	SPIN
5	93	91	99	96	1.24	2.20
6	117	115	123	120	2.38	9.50
7	143	141	149	146	2.44	39.40
8	171	169	177	174	3.13	(97.30)[3]
9	201	199	207	204	3.68	
10	233	231	239	236	5.01	

Table 1. Results and comparison with SPIN for Chang and Roberts' algorithm.

A snapshot algorithm Consider a distributed system with N processes and one single monitor process M. Every process can synchronously communicate with its neighbour processes and with the monitor process. The task of a snapshot algorithm is to enable any process at any time to initiate a snapshot that is generated in the monitor process M. After the generation of a single snapshot all processes receive it and they are reinitialized.

We have implemented Bouge's snapshot algorithm in B(PN)^2 for a ring architecture of 4 processes.

Verification and results The task of the snapshot algorithm can be specified by the following LTL-formula[4]:

$$\Box \left(\left(\bigvee_{i=0}^{3} active_i = \mathbf{true} \right) \Rightarrow \diamond snapshot\_generated = \mathbf{true} \right)$$

The Petri net corresponding to the B(PN)^2-program has 175 places and 178 transitions. The product net contains 179 places, 178 transitions, and 254 different[5]

[3] 128 Mbytes main memory are exceeded.

[4] Here, $active_i$ denotes the local variable of the i-th process.

[5] Different w.r.t. their support.

final semipositive T-invariants. The P-components used for the T^*-invariant test were those corresponding to the variables of the $B(PN)^2$-program. The product net was constructed in 81 seconds, and the absence of T^*-invariants was checked in 64 seconds.

This example could not[6] be verified by SPIN. It could not be verified by the stubborn set method either. We tried to compute the stubborn reduced reachability graph using Starke's INA tool, but had to abort the process after 20 hours, when 206000 reduced states had been generated.

7 Conclusions

We have presented a semidecision test for the model-checking problem of 1-safe Petri nets and LTL. The model-checking problem is first reduced to the emptiness problem of a Büchi net. Then, the test checks the presence or absence of a particular class of T-invariants which we have called T^*-invariants. If no T^*-invariants are present, then the Büchi net is empty, and the property holds. We were able to implement this check very easily by making use of the constraint programming tool 2lp. We have shown that there exist real algorithms for which our test allows to verify a property which cannot be proved using other exact methods.

We finish the section with some comments:

On techniques for emptiness checking. Emptiness of Büchi nets can also be checked using exact methods, not only semidecision tests. Wallner is working on the application of net unfoldings to this problem [17].

On the restriction to 1-safe Petri nets. In the paper we have restricted our attention to 1-safe Petri nets. A different version of our test, however, can also be applied to arbitrary Petri nets, even unbounded ones (which is not true of the automata-theoretic approach). Essentially, instead of T-invariants it is necessary to work with so called *T-surinvariants*.

On the T^*-invariant test. The test we have developed is certainly not the only possible one. We see it more as an experiment in using structural information to prove liveness properties of real examples. We have implemented some such tests in the PEP-tool, which can be applied when exact methods fail.

On the complexity of the test. It may be criticized that our test involves solving an NP-complete problem (absence of T^*-invariants), which may require exponential time. Actually, we think that good tests *are likely to be* NP-complete. Complexity results show that nearly all interesting verification problems about 1-safe Petri nets are PSPACE-complete. Polynomial tests for such problems are bound to have poor quality, as confirmed by our experiments. NP-complete tests lie between the poor quality polynomial tests and the PSPACE-complete exact methods.

On the 2lp implementation. Constraint programming tools like 2lp open a wide range of new possibilities in the application of structural objects like invariants, siphons and traps to verification problems. They also allow to implement

[6] 128 Mbytes main memory are exceeded.

prototypes very quickly.

Acknowledgements We wish to thank Robert Riemann for his critical comments on an earlier version of this paper. We also benefited from discussions with Frank Wallner and Ahmed Bouajjani.

References

1. Eike Best, Raymond Devillers, and Jon G. Hall. *The Box Calculus: a New Causal Algebra with Multi-Label Communication.* Number 4/92 in Hildesheimer Informatik-Bericht. Universität Hildesheim, Mai 1992.
2. Eike Best and Richard Pinder Hopkins. $B(PN)^2$ – a Basic Petri Net Programming Notation. In A. Bode, M. Reeve, and G. Wolf, editors, *Proceedings of PARLE '93*, volume 694 of *Lecture Notes in Computer Science*, pages 379 – 390, 1993.
3. Luc Bouge. Repeated Synchronous Snapshots and their Implementation in CSP. In W. Brauer, editor, *Proceedings 12th ICALP*, volume 194 of *Lecture Notes in Computer Science*, pages 63 – 70. Springer, 1981.
4. Ernest Chang and Rosemary Roberts. An Improved Algorithm for Decentralised Extrema-finding in Circular Distributed Systems. *Communication of the ACM*, 22(5):281 – 283, 1979.
5. Jörg Desel and Javier Esparza. *Free Choice Petri Nets.* Cambridge University Press, 1995.
6. J. Esparza and S. Melzer. Model-Checking LTL using Constraint Programming. Technical report, Technische Universität München, March 1997. Available at http://papa.informatik.tu-muenchen.de/forschung/sfb342_a3/refs.html.
7. Javier Esparza and Glenn Bruns. Trapping Mutual Exclusion in the Box Calculus. *Theoretical Computer Science*, 153:95 – 128, 1996.
8. Rob Gerth, Doron Peled, Moshe Vardi, and Pierre Wolper. Simple On-the-fly Automatic Verification of Linear Temporal Logic. In *Protocol Specification Testing and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall.
9. Patrice Godefroid. *Partial-Order Methods for Verification of Concurrent Systems*, volume 1032 of *Lecture Notes in Computer Science*. Springer, 1996.
10. Bernd Grahlman and Eike Best. PEP – More than a Petri Net Tool. In T. Margaria and B. Steffen, editors, *TACAS '96*, volume 1055 of *Lecture Notes in Computer Science*, pages 397 – 401. Springer-Verlag, 1996.
11. Gerald J. Holzmann. *Basic Spin Manual.* AT&T Bell Lab., Murray Hill.
12. Joxan Jaffar and Jean-Lois Lassez. Constraint logic programming. In *14th Annual ACM Symposium on Principles of Programming Languages*, 1987.
13. Ken McAloon and Carol Tretkoff. *Optimization and Computational Logic.* John Wiley & Sons, 1996.
14. Antti Valmari. A Stubborn Attack on State Explosion. *Formal Methods in System Design*, 1:297 – 322, 1992.
15. M. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1 – 37, 1994.
16. Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logics in Computer Science*, pages 322 – 331, Cambridge, June 1986.
17. F. Wallner. Model-Checking LTL using Net Unfoldings. Technical report, Technische Universität München, Institut für Informatik, Forthcoming 1997.