LTL Model Checking for Modular Petri Nets

Timo Latvala* and Marko Mäkelä

Laboratory for Theoretical Computer Science
Helsinki University of Technology
P.O. Box 5400
FIN-02015 HUT
Finland
`{Timo.Latvala, Marko.Makela}@hut.fi`

Abstract. We consider the problem of model checking modular Petri nets for the linear time logic LTL-X. An algorithm is presented which can use the synchronisation graph from modular analysis as presented by Christensen and Petrucci and perform LTL-X model checking. We have implemented our method in the reachability analyser Maria and performed experiments. As is the case for modular analysis in general, in some cases the gains can be considerable while in other cases the gain is negligible.

Keywords: Modular Petri nets, LTL-X, model checking, Maria.

1 Introduction

Modelling using Petri nets can be made easier in many ways. Examples of extensions of simple Place/Transition nets which ease modelling include, adding types as in Coloured Petri Nets [1], or allowing modular specifications as in [2]. If, however, we are to reap the full benefits of easier modelling, analysis methods must also scale up to take advantage of the new features.

One of the most powerful methods of analysing the behaviour of a Petri net is *reachability analysis*. By constructing the set of reachable markings we can decide important properties such as if the net is live, does a certain invariant hold for all states, etc. Perhaps the most flexible method of analysis we can use is *model checking* [3]. Model checking allows us to check if the behaviour of the net corresponds to a specification given in a temporal logic such as the linear-time temporal logic (LTL).

Modular Petri nets as presented by Christensen and Petrucci [2] allow designers to specify a system as communicating modules. Modules communicate using shared transitions or fusion places. Although one of the chief motivations for using modular specifications is to ease the design of complex systems, another reason is facilitating *compositional reasoning*. All analysis methods suffer from the so called state explosion problem (see, e.g., [4]). Compositional analysis tries

to alleviate the problem by considering modules in isolation and then reason about the system as a whole.

Christensen and Petrucci [2] presented a way to perform invariant analysis, reachability analysis and how to prove several important properties for modular nets in a modular way. Modular reachability analysis works by hiding internal moves of the modules. Mäkelä [5] extended their approach for reachability analysis and implemented modular reachability analysis for hierarchical modular High-level nets. The work also covers model checking of safety properties using the translation from LTL safety properties to finite automata implemented in [6]. Full LTL model checking is something which has been missing.

In this work we show how model checking for the temporal logic LTL-X of the synchronisation graph produced by modular analysis can be achieved. We have implemented our method in the reachability analyser Maria [7] and present experimental results. Our results indicate that the overhead of the method is fairly small, which means that the method works well when modular analysis is efficient. Our work is inspired by a somewhat similar LTL model checking method for unfoldings of Petri nets [8]. There are also similarities with the work done on testers [9,10].

There are many methods which try to use some form of compositional analysis. As summarised in the survey article [11], using Kronecker algebra, especially with stochastic Petri nets, can allow analysis of the system as whole using the components. Compositional reasoning as described by Valmari [12] advocates the use of process algebraic equivalences for minimisation and composition of components, to form an equivalent smaller system where properties can be proved easily. Another method of compositional reasoning is to use assume guarantee reasoning (see, e.g., [13]). The basic idea is to prove properties of the modules separately and then conclude that the wanted property holds for the global system.

2   Definitions

Definition 1. *A Place/Transition net (PT-net) is a tuple $N = (P, T, W, M_0)$ where,*

- *P is a finite set of places,*
- *T is a finite set of transitions such that $P \cap T = \emptyset$,*
- *$W : (P \times T) \cup (T \times P) \to \mathbb{N}$ is the arc weight function,*
- *$M_0 : P \to \mathbb{N}$ is the is initial marking.*

A marking is a multiset over P. For a transition $t \in T$ we identify $t^{\bullet}$ ($^{\bullet}t$) with the multiset given by $t^{\bullet}(p) = W(t, p)$ ($^{\bullet}t(p) = W(p, t)$) for any $p \in P$.

A transition $t \in T$ is *enabled* in a marking M iff $^{\bullet}t \subseteq M$. A transition t enabled in a marking M can *occur* resulting in the marking $M' = M - {}^{\bullet}t + t^{\bullet}$. This is denoted $M \xrightarrow{t} M'$. A marking in which no transition is enabled is called a *deadlocking* marking.

An *execution* of a net is an infinite sequence of markings $\xi = M_0 M_1 M_2 \ldots$ such that $M_0 \xrightarrow{t_0} M_1 \xrightarrow{t_1} \ldots$. The corresponding *trace* is the infinite sequence of transitions $\sigma = t_0 t_1 t_2 \ldots$. A finite sequence $M_0 \xrightarrow{t_0} M_1 \xrightarrow{t_1} \ldots \xrightarrow{t_{n-1}} M_n$, where M_n is a deadlocking marking, can be seen as an execution by repeating the last marking forever, i.e., $M_0 M_1 \ldots M_n M_n M_n \ldots$.

Definition 2. *The* reachability graph $G = (V, E, v_0)$ *of a net* $N = (P, T, W, M_0)$ *is defined inductively as follows:*

- $v_0 = M_0 \in V$.
- *If* $M \in V$ *and* $M \xrightarrow{t} M'$ *then* $M' \in V$ *and* $(M, t, M') \in E$.
- V *and* E *contain no other elements.*

The reachability graph of a net describes the dynamic behaviour of the net. In general, the reachability graph can be infinite but in this work we will assume it is finite unless explicitly stated otherwise. Many properties of a net, such as home markings and deadlocks, can be decided in linear time w.r.t. the size of the reachability graph. A simple PT-net modelling a mutual exclusion algorithm and its reachability graph can be found in Figure 1. The model shows two processes (l and r) competing for a critical section which is guarded by a lock. A quick inspection of the reachability graph confirms that under no circumstances are both processes in the critical section at the same time.
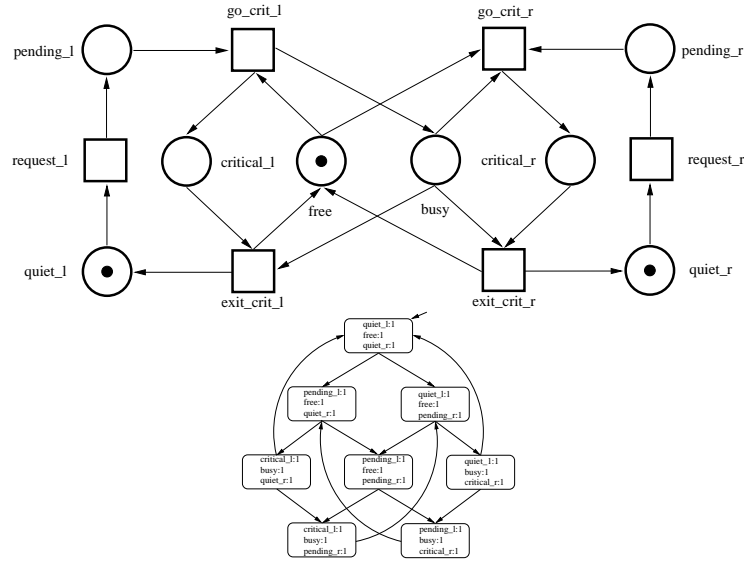
Fig. 1. A simple model of a mutual exclusion algorithm and its reachability graph.

Although PT-nets have great modelling power, the lack of structure can sometimes be a problem. It is conceptually easier to deal with large systems as modules, because it allows the designer to consider different parts of the system in relative isolation. Furthermore, the structural information can in some cases be utilised to reduce the complexity of analysis.

Modular PT-nets introduce structure to PT-nets by letting modules be specified separately. The modules communicate either by using shared transitions or place fusion. We restrict ourselves to nets which communicate using shared transitions. Christensen and Petrucci [2] have shown that modular nets with place fusion can be transformed to nets using only shared transitions.

Definition 3. *A modular PT-net is tuple* $\Sigma = (S, TF)$ *where:*

- S *is a finite set of* modules*:*
  - *each module* $s \in S$ *is a PT-net* $s = (P_s, T_s, W_s, M_{0_s})$,
  - *the sets of nodes corresponding to different modules are pairwise disjoint, i.e., for all* $s_1, s_2 \in S : s_1 \neq s_2 \Rightarrow (P_{s_1} \cup T_{s_1}) \cap (P_{s_2} \cup T_{s_2}) = \emptyset$.
- *Let* $T = \bigcup_{s \in S} T_s$ *be the set of all transitions.* $TF \subseteq 2^T$ *is a finite set of* transition fusion sets *such that for all* $tf \in TF$ *we have that if* $t_i, t_j \in tf$ *and* $i \neq j$ *then* $t_i \in T_s \Rightarrow t_j \notin T_s$. *In other words, a module may contribute only one transition to a fusion transition.*

Transition fusion sets model synchronising actions. Because the nodes of the modules are pairwise disjoint, the global marking of a modular net is simply the union of the markings of the modules. In this work, we mostly use the global marking of the net and simply denote it M as before. We denote by $ET \subseteq T$ the set of transitions which belong to a transition fusion set and by $IT = T \setminus ET$ the set of internal transitions.

Since there are both fusion transitions and internal transitions in a modular net, we need a uniform way to refer to them. Here, we call them transition groups and they are essential equivalent to the transition concept in a standard PT-net.

Definition 4. *A* transition group $tg \subseteq T$ *is a set of transitions such that it consists of a single internal transition* $t \in IT$ *or is equal to a transition fusion set* $tf \in TF$. *The set of all transition groups is denoted* TG.

We extend the preset and postset notation to transition groups. Let ${}^{\bullet}tg$ denote the multiset given by

$$ {}^{\bullet}tg(p) = \sum_{t \in tg} W(t, p) \qquad \text{where } W = \bigcup_{s \in S} W_s $$

The notation for the postset of a transition group is generalised in a similar manner. With this notation, enabledness for a transition group generalises in a natural way.

Definition 5. *A transition group* $tg \in TG$ *is* enabled *in a marking* M *if*

$$ {}^{\bullet}tg \subseteq M $$

The result of occurrence of a fusion transition generalises as expected. An enabled transition group tg can occur in marking M resulting in a marking M' given by

$$M' = M - {}^\bullet tg + tg^\bullet$$

It is useful to differentiate between the firing of an internal or an external transition. We therefore introduce the following notation.

- $M[[t\rangle M'$ denotes that M' is reachable from M by firing an internal transition.
- $M[tf\rangle\rangle M'$ denotes that M' is reachable from M by firing a fusion transition tf.
- $M[[\sigma\rangle\rangle M'$, where $\sigma = t_0 t_1 t_2 \ldots t_n tf$, denotes that M' is reachable from M by a sequence of internal transitions followed by a fused transition.

In Figure 2 we show the same mutual exclusion algorithm as in Figure 1 modelled as a modular net. We have split the net into three modules. One module each for the competing processes and one module for the lock. The fusion sets are indicated by the labels on the transitions. Note that the transitions 'lock' and 'unlock' belong to several transition fusion sets.
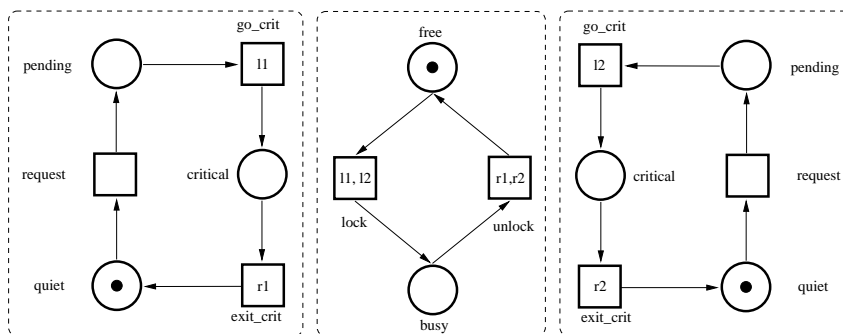

Fig. 2. A mutual exclusion algorithm as a modular net.

3   Modular Reachability Analysis

One of the chief motivations for using modular Petri nets is the possibility of using modular analysis [2]. With modular analysis we can analyse the behaviour of the net without explicitly constructing the full reachability graph. Instead we only construct the so called synchronisation graph between the modules. The synchronisation graph only includes the external moves, i.e., moves where several modules participate—all internal moves are hidden. For loosely coupled systems the synchronisation graph can be significantly smaller than the full reachability graph.

The key idea behind modular analysis is having a single marking represent all markings which can be reached from that marking with internal transitions. We define $\Pi(M)$ to be the set of markings reachable from M using a possibly empty sequence of internal transitions. Two markings M, M' are considered equal iff $\Pi(M) = \Pi(M')$.

Definition 6. *Let $\Sigma = (S, TF, M_0)$ be a modular net. The* synchronisation graph $\mathbf{G} = (\mathbf{V}, \mathbf{E}, \mathbf{v}_0)$ *of the net is defined inductively as follows:*

- $\mathbf{v}_0 = M_0 \in \mathbf{V}$.
- *If $M \in \mathbf{V}$ and $\exists M' \in \Pi(M) : M'[tf\rangle M''$ then $(M, tf, M'') \in \mathbf{E}$ and $M'' \in \mathbf{V}$.*
- $\mathbf{V}$ *and* $\mathbf{E}$ *contain no other elements.*

The second item of the definition describes which markings are stored in the graph. The successor markings of a global marking M can be computed, e.g., by exploring in each module the set of local markings reachable from M via internal transitions and recording where there are enabled external transitions, and then composing global markings and firing the corresponding transition fusion sets [5, Section 4.1].

Christensen and Petrucci [2] describe how the synchronisation graph can be computed for modular Petri nets. Their approach was extended to hierarchical high-level nets by Mäkelä [5], who also considered verification of simple safety properties.

In Figure 3 we show the synchronisation graph of the modular net given in Figure 2. The synchronisation graph has three states and four arcs compared to original reachability graph which has eight states and fourteen arcs. The arcs are labelled by the fusion sets given in the net description. From the graph it is easy see that the mutual exclusion property holds. What the graph does not show is the interleaving between the processes when they change from quiet to pending.
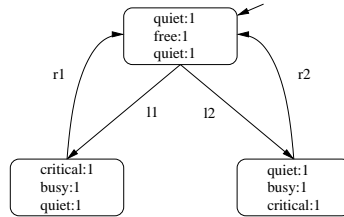
Fig. 3. The synchronisation graph for the mutual exclusion algorithm.

4   Model Checking LTL-X

Model checking the synchronisation graph would be interesting as it can be significantly smaller than the reachability graph. However, it is not immediately clear how the synchronisation graph should be used in model checking. The traditional automata-theoretic solution [14,15,16], to synchronise the graph with a Büchi automaton representing the given LTL formula is not directly applicable, as the synchronisation graph hides information. However, by considering what information is preserved by the synchronisation graph an automata-theoretic model checking method can be devised.

An LTL formula φ is defined over a set of atomic propositions AP. The models of the formula are infinite words over 2^{AP}. An LTL formula has the following syntax:

1. $\psi \in AP$ is an LTL formula.
2. If ψ and φ are LTL formulae then so are $\neg\psi$, $\mathbf{X}\psi$, $\psi\,\mathbf{U}\,\varphi$ and $\psi\vee\varphi$.

We denote the suffix of a model $\pi = \sigma_0\sigma_1\sigma_2\ldots \in (2^{AP})^\omega$ by $\pi^i = \sigma_i\sigma_{i+1}\sigma_{i+2}\ldots$. The semantics of LTL are inductively defined using the 'models' relation $\models$:

- $\pi^i \models \psi$ iff $\psi \in \sigma_i$ for $\psi \in AP$.
- $\pi^i \models \neg\psi$ iff $\pi \not\models \psi$.
- $\pi^i \models \psi \vee \varphi$ iff $\pi \models \psi$ or $\pi \models \varphi$.
- $\pi^i \models \mathbf{X}\psi$ iff $\pi^{i+1} \models \psi$.
- $\pi^i \models \psi\,\mathbf{U}\,\varphi$ iff $\exists k \geq i$ such that $\pi^k \models \varphi$ and $\pi^j \models \psi$ for all $i \leq j < k$.

If $\pi^0 \models \psi$ we simply write $\pi \models \psi$. Common abbreviations used are $\top = p \vee \neg p$ for some arbitrary $p \in AP$, the usual abbreviations for the Boolean operators $\wedge, \Rightarrow$ and $\Leftrightarrow$, and the temporal operators 'finally' $\mathbf{F}\psi \equiv \top\,\mathbf{U}\,\psi$ and 'globally' $\mathbf{G}\psi \equiv \neg\mathbf{F}\neg\psi$.

The subset of LTL where the 'next' operator $\mathbf{X}$ is not allowed is denoted LTL-X. The 'next' operator allows LTL to specify properties which can differentiate between sequences which only have different internal moves, i.e., moves which do not affect the truth of relevant atomic propositions, or so called *stuttering*. Formally, two models $\pi = \sigma_0\sigma_1\ldots, \pi' = \sigma'_0\sigma'_1\ldots$ are stuttering equivalent if there are two infinite sequences of positive integers $0 = i_0 < i_1 < i_2 < \ldots$ and $0 = j_0 < j_1 < j_2 < \ldots$ such that for every $k \geq 0$: $\sigma_{i_k} = \sigma_{i_k+1} = \cdots = \sigma_{i_{k+1}-1} = \sigma'_{j_k} = \sigma'_{j_k+1} = \cdots = \sigma'_{j_{k+1}-1}$.

It is a well-known fact that LTL-X is insensitive to stuttering (see, e.g., [17]). Because the synchronisation graph hides internal moves of the modules, the sequences generated by the synchronisation graph can differ from sequences generated by the reachability graph by stuttering. We therefore focus on model checking for LTL-X.

A formula defines a language $\mathcal{L}(\varphi) = \{w \in (2^{AP})^\omega \mid w \models \varphi\}$. The language of the LTL formula can be captured by a *Büchi automaton* (see, e.g., [18]). In the recent years several papers have dealt with the problem of translating an LTL formula to a Büchi automaton [19,20,21].

Definition 7. *A Büchi automaton is a tuple $\mathcal{A} = (Q, A, \rho, q_0, Q_F)$, where Q is a finite set of states, A is a finite alphabet, $\rho \subseteq Q \times A \times Q$ is the transition relation, $q_0 \in Q$ is the initial state, and $Q_F \subseteq Q$ is a set of accepting states.*

An infinite word $w \in A^\omega$ generates a run $r = q_0 q_1 q_2 \ldots$ of the automaton such that q_0 is the initial state of the automaton and for all $i \geq 0$ we have $(q_i, w(i), q_{i+1}) \in \rho$. If the automaton is non-deterministic, one word can generate several runs. A word w is accepted iff it has run $r : \mathbb{N} \to Q$ such that $r(i) \in Q_F$ for infinitely many i. We use $\mathcal{A}^q$ to denote the automaton $\mathcal{A}$ with q set as the initial state. The language of the automaton, denoted $\mathcal{L}(\mathcal{A})$, is defined as the set of strings accepted by the automaton. Here we assume that when Büchi automata are used to represent LTL formulae the alphabet is $A = 2^{2^{AP}}$.

Definition 8. *Given a marking M, the function $eval(M)$ returns the set of atomic propositions which hold in M. The notation is extended to sequences of markings in the normal way.*

For a Petri net N, we write $N \models \psi$ iff for all executions ξ of the net, we have that $eval(\xi) \models \psi$. The executions of the Petri net define a language when projected with the $eval$ function. A Petri net has a given temporal property if the language of the net, as defined above, is a subset of the language of the property automaton.

The traditional way of model checking a Petri net using the automata theoretic approach, is to check if intersection of the language of the Petri net with the language of the *negation* of the property is empty. This uses the fact that for any two languages L_1, L_2 the equivalence $L_1 \subseteq L_2 \Leftrightarrow L_1 \cap \overline{L_2} = \emptyset$ holds, which is why it is referred to making an emptiness check. The intersection of the languages is computed by synchronising the reachability graph of the net with a Büchi automaton representing the negation of the property. If the synchronisation has no accepting run, the property holds.

The usual way of computing the intersection with automaton and the reachability graph requires that the Büchi automaton synchronises with every move of the net. Because modular analysis relies on hiding the internal moves of the modules this is not a good approach, because it would make all moves external and forfeit the potential benefit of using modular analysis.

It is, however, possible to do model checking by only synchronising with the visible transitions. The price we pay is a more complex model checking algorithm. The approach we present is similar to the work on model checking using unfoldings of Petri nets [8] and has common elements with the tester approach [9] but as we are synchronising with different constructs there are technical differences.

Let φ be a formula over a set AP of atomic propositions (Boolean expressions on markings of the net). We call a place $p \in P$ *visible* if the truth of an atomic proposition can be changed by altering the marking of the place. Let $P_V \subseteq P = \bigcup_{s \in S} P_s$ be the set of visible places. The set of visible transitions is defined as $T_V = {}^\bullet P_V \cup P_V^\bullet$. Because all changes in atomic proposition must be visible in the

synchronisation graph we require that $T_V \subseteq ET$ (in an implementation we could automatically detect which transition need to be treated as external transitions).

Our goal is to find the illegal executions of the modular net by synchronising the Büchi automaton with the *visible* transitions. Formally, we define the synchronisation in the following way. Let $\mathcal{A}_{\neg\varphi}$ be a Büchi automaton accepting the language of the negation of the property φ and Σ a modular net. If the net is in the marking M, the automaton is in the state q, and $(q, a, q') \in \rho$ such that $eval(M) \in a$, then the net and the automaton will synchronise in the following way:

- All visible transitions must *always* be synchronised with the Büchi automaton. If a visible transition $t \in T_V$ is enabled in $\Pi(M)$, it is synchronised with the automaton, and the product moves to the state (M', q') where $M[tf\rangle\rangle M'$.
- Invisible transitions can occur in the system without synchronising with the automaton.

A state in the synchronisation $s = (M, q)$ is accepting if $q \in Q_F$. The synchronisation state (M, q) belongs to a livelock set I if $\mathcal{A}^q_{\neg\varphi}$ accepts $eval(M)^\omega$. The livelock set I can be very large and should thus be computed on demand when model checking.

We say that the net has an illegal ω-execution if there is an execution of the synchronisation where the corresponding trace has infinitely many visible transitions and there are infinitely many occurrences of an accepting state in the execution. The net has an illegal livelock if there is an execution of the synchronisation $s_0 s_1 s_2 \ldots s_n s_{n+1} \ldots$ such that $s_n \in I$ and the corresponding trace from s_n onward $\sigma^n = t_n t_{n+1} t_{n+2} \ldots$ does not contain any visible transition.

We can detect all illegal ω-executions and illegal livelocks by computing a synchronised product of the Büchi automaton and the synchronisation graph of the modular net.

Definition 9. *Let $\mathcal{A}_{\neg\psi} = (Q, A, \rho, q_0, Q_F)$ be a Büchi automaton and $\mathbf{G} = (\mathbf{V}, \mathbf{E}, \mathbf{v}_0)$ be a synchronisation graph. Their* product (V_p, E_p, p_0, F, I) *is defined in the following way:*

- *$p_0 = (\mathbf{v}_0, q_0) \in V_p$.*
- *Given $(M, q) \in V_p$, $(M, t, M') \in \mathbf{E}$, and $(q, a, q') \in \rho$ there are two possibilities:*
    - *If $t \in T_V$ and $eval(M) \in a$, the net and the automaton synchronise and we have $((M, q), \{t, a\}, (M', q')) \in E_p$ and $(M', q')' \in V_p$.*
    - *If $t \notin T_V$, the system moves: $((M, q), t, (M', q)) \in E_p$ and $(M', q) \in V_p$.*
- *V_p and E_p contain no other elements.*
- *$F = \{(M, q) \in V_p \| q \in Q_F\}$.*
- *$I = \{(M, q) \in V_p \| \mathcal{A}^q_{\neg\psi} \text{ accepts } eval(M)^\omega\}$.*

We claim that any execution of the net which breaks the given LTL specification will induce either an illegal ω-execution or an illegal livelock, which will also show up in the product of the synchronisation graph and the Büchi automaton.

Theorem 1. *Given a modular net Σ and a Büchi automaton $\mathcal{A}_{\neg\psi}$, $\Sigma \not\models \psi$ iff the product of the automaton and the synchronisation graph of the net has an illegal ω-trace or an illegal livelock.*

The proof is fairly similar to the proof of Theorem 2 given in [22] but as there are some technical differences we present it here.

Proof. Let $\xi = M_0 M_1 M_2 \ldots$ be an execution of the net Σ such that $eval(\xi) \not\models \psi$. By construction, we know that $\mathcal{A}_{\neg\psi}$ accepts $w = eval(\xi)$. Let $w' = eval'(\xi)$ be the same sequence with all stuttering removed. There are now two possible cases: w' can either be an infinite (a) or a finite sequence (b).

(a) Because all properties specified by LTL-X are immune to stuttering, $\mathcal{A}_{\neg\psi}$ accepts the infinite w'. Let $r' = q_0 q_1 q_2 \ldots$ be one of the runs accepting w'. The product has the following illegal ω-execution. Set (M_0, q_0) as the initial state. Set $j = 0$. For each $i \geq 0$ do: (i) fire the transition t_i which leads to M_{i+1}, (ii) if $t_i \in IT$ then goto (i), otherwise if $t_i \in T_V$ set $j = j + 1$ (iv) (M_{i+1}, q_j) is the next state in the run. This sequence will exist as the synchronisation graph contains all possible visible sequences. Step (ii) deals with invisible internal transitions which are not present in the synchronisation graph while (iii) makes sure that the Büchi automaton advances only when we have a visible external transition. Because w' is an accepted sequence, the run r' has a final state occurring infinitely often and thus the product has an illegal ω-trace.

(b) In the same manner as in (a) we can argue that the finite w' will induce a finite run of the product. Let (M_i, q_i) be the final state of this run. Because $\mathcal{A}_{\neg\psi}$ accepts the full word w, we know that w' can be extended by stuttering to an accepting word. Thus, by the definition of I we know that $(M_i, q_i) \in I$. Since ξ was infinite we know it is possible to fire an infinite sequence of invisible transitions from M_i onward and consequently the product has an illegal livelock.

Let ξ_p be an illegal ω-execution of the product. By projecting the Büchi component of the run onto $\mathcal{A}_{\neg\psi}$ it is clear that this is an accepting run for the automaton. Similarly we can easily build a run of Σ from the net component of the execution. All it requires is finding the fired internal transition which occur between the external transitions. This is be possible due to the properties of the synchronisation graph. Essentially, we only need to compute how to enable the next fusion transition by firing internal transitions. By the properties of $\mathcal{A}_{\neg\psi}$ we can then conclude that $\Sigma \not\models \psi$.

Let ξ_p be an illegal livelock of the product and (M_i, q_i) be the state after which only invisible transitions are executed. We can project the Büchi component onto $\mathcal{A}_{\neg\psi}$ such that the trace ends in q_i. We know that (M_i, q_i) is in I and thus by implication that $\mathcal{A}_{\neg\psi}^q$ will accept $eval(M_i)^\omega$. The loop of invisible transitions corresponds to this infinite stuttering. Building an infinite execution of Σ from ξ_p is again easy. Thus we can again conclude that $\Sigma \not\models \psi$. $\square$

Finding an illegal livelock or an illegal infinite trace from the product is equivalent to finding an error specified by a tester as described by Valmari [9]. As suggested in [10], we then have a solution which traverses the product three times in the worst case by first using Valmari's one-pass algorithm [9] to find any illegal livelocks. If no illegal livelocks are found we can use the standard nested depth-first algorithm [23] to find any illegal infinite traces. Some small modifications are required for the algorithms to function correctly. For the one-pass algorithm we need an efficient way of deciding if a state belongs the set I. This corresponds to model checking a reachability graph consisting of a single marking with a self loop. If a state M belongs to the set I, not only must we check if there is a loop of invisible transitions in the product starting from M, but also if internal transitions can loop in any of the modules. This can be implemented by depth-first traversal of the states reachable from the local states of the modules corresponding to M. Figure 4 describes the model checking algorithm at an abstract level.

Input: The product (V_p, E_p, p_0, F, I)
proc model check(V_p, E_p, p_0, F, I) **begin**
    for all states $(M, q) \in V_p$ **do**
        if $(M, q) \in I$ **then**
            if a loop of invisible transitions starts from M **then**
                return "illegal livelock found"
        fi
        if $(M, q) \in F$ **then**
            if (M, q) is reachable from (M, q) in the product **then**
                return "illegal ω-execution found"
        fi
    od
end

Fig. 4. Abstract algorithm for model checking.

5   Experiments

We have implemented our method in the reachability analyser Maria [7]. In order to evaluate our implementation we have conducted some experiments. We compared modular model checking against the basic Maria model checker [24]. Additionally, we also ran a few benchmarks against PROD [25], a tool with advanced partial order reduction methods. This benchmark gives us some indication on how modular analysis compares with another method producing stuttering equivalent structures.

We used three different models of which two were parametric. The first model (AGV) describes a system of automated guided vehicles, first modelled by

Petrucci [26]. The second model (SW) is a variant of a sliding window protocol and the third one (LE) models the leader election protocol in a unidirectional ring [27]. The results of the experiments can be found in Table 1. The statistics we recorded were number of states and arcs in the reachability graph, number of states in the product, time used for state space construction and model checking, size of the synchronisation graph, number of states in the product, the time used, and the type of formula. All tests were run on a machine with 1 GB of RAM with an AMD Athlon XP 2000 processor.

Table 1. Experimental results.

Sys-tem	Flat state space $G = (V, E, v_0)$				Modular $\mathbf{G} = (\mathbf{V}, \mathbf{E}, \mathbf{v}_0)$				ψ
	$|V|$	$|E|$	product	time/s	$|\mathbf{V}|$	$|\mathbf{E}|$	product	time/s	
AGV	30,965,760	216,489,984	N/A	N/A	87,480	464,616	87,492	27.3	$\mathbf{GF}\varphi$
SW$_4$	6,360	16,608	14,857	1.3	4,456	16,016	8,889	2.6	$\mathbf{GF}\varphi$
SW$_5$	24,270	68,760	52,891	5.8	16,930	72,660	31,991	13.1	$\mathbf{GF}\varphi$
SW$_6$	82,884	248,400	169,645	20.6	57,564	286,488	103,477	118	$\mathbf{GF}\varphi$
LE$_3$	159	303	314	0.0	35	65	68	0.0	$\mathbf{FG}\varphi$
LE$_4$	716	1,851	1,428	0.2	92	229	182	0.1	$\mathbf{FG}\varphi$
LE$_5$	3,432	11,198	6,860	1.3	253	802	504	0.2	$\mathbf{FG}\varphi$
LE$_6$	16,792	66,043	33,580	8.0	715	2,748	1,428	0.8	$\mathbf{FG}\varphi$
LE$_7$	82,667	380,267	165,330	49.3	2,043	9,212	4,084	2.9	$\mathbf{FG}\varphi$
LE$_8$	407,699	2,146,965	815,394	295	5,865	30,308	11,728	10.2	$\mathbf{FG}\varphi$

It would appear that in some cases the modular algorithm is faster while in some cases it is slower. For very loosely coupled models as AGV, the modular algorithm does well. Analysis of a model with a fair amount of synchronisation, such as LE, also shows gains using our modular algorithm. When the gains of using modular analysis are questionable as for the SW models, the overhead of using modular analysis and the modular algorithm for model checking is significant but not prohibitively expensive.

Our benchmarks against PROD were conducted with a special version of the sliding window protocol model with more realistic timeout conditions. The model was separately optimised for PROD and Maria in order to make the comparison as fair as possible. Results can be found from Table 2.

Table 2. Benchmarks against PROD.

Sys-tem	PROD $G = (V, E, v_0)$				Modular (Maria) $\mathbf{G} = (\mathbf{V}, \mathbf{E}, \mathbf{v}_0)$				ψ
	$|V|$	$|E|$	product	time/s	$|\mathbf{V}|$	$|\mathbf{E}|$	product	time/s	
SW$_{2,2}$	8,384	13,388	17,622	8.0	7,376	48,860	9,709	8.0	$\mathbf{GF}\varphi$
SW$_{3,2}$	131,555	198,466	270,142	245	86,995	802,650	101,551	148	$\mathbf{GF}\varphi$
SW$_{3,3}$	422,484	590,298	859,724	969	267,192	2,885,022	302,551	757	$\mathbf{GF}\varphi$
SW$_{4,2}$	1,434,750	2,056,176	2,914,484	7556	762,870	9,379,788	836,275	3031	$\mathbf{GF}\varphi$

The results seem to indicate that the number of states produced is fairly similar while partial order reductions eliminate arcs much more successfully. Modular analysis produces results faster but the difference is not large. The more relaxed synchronisation for computing the product in modular analysis can obviously lead to smaller products. A combination of modular analysis and partial order reductions could produce good results and would be very interesting.

6 Conclusions

In this paper we have shown how LTL-X model checking can be done on the synchronisation graph resulting from modular analysis of modular Petri nets as presented in [2]. Our method requires a different form of synchronisation compared to the traditional automata theoretic model checking and a somewhat more complicated emptiness checking algorithm. The time complexity overhead is, however, only linear compared to conventional emptiness checking algorithms for Büchi automata.

As the model checking algorithm has a reasonable overhead (not much worse than the traditional algorithm), the performance of model checking for modular nets is heavily dependent on how well modular analysis performs. This means that for many models where modular analysis does not reduce the state explosion, compared to reachability analysis of flat models, using our method will not result in excessive waiting. Our implementation is available as a patch to the standard Maria distribution from `http://www.tcs.hut.fi/Software/maria/`.

As future work we are considering refining the concept of visibility. A more relaxed view of visibility could potentially improve the performance of the model checking algorithm. In general, we believe that the efficiency of modular analysis could be improved by developing partial order methods which are compatible with modular analysis. The two problems are related as visibility is also an issue in partial order reductions.

Acknowledgements

References

1. Jensen, K.: Coloured Petri Nets. Volume 1. Springer, Berlin (1997)
2. Christensen, S., Petrucci, L.: Modular analysis of Petri Nets. The Computer Journal **43** (2000) 224–242
3. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge, Massachusetts (1999)
4. Valmari, A.: The state explosion problem. In: Lectures on Petri Nets I: Basic Models. Volume 1491 of LNCS. Springer (1998) 429–528
5. Mäkelä, M.: Model checking safety properties in modular high-level nets. In: Application and Theory of Petri Nets (ICATPN'2003). Volume 2679 of LNCS., Springer (2003) 201–220

6. Latvala, T.: Efficient model checking of safety properties. In Ball, T., Rajamani, S., eds.: Model Checking Software. 10th International SPIN Workshop. Volume 2648 of LNCS., Springer (2003) 74–88

7. Mäkelä, M.: Maria: modular reachability analyser for algebraic system nets. In Esparza, J., Lakos, C., eds.: Application and Theory of Petri Nets 2002. Volume 2360 of LNCS., Springer (2002) 434–444

8. Esparza, J., Heljanko, K.: Implementing LTL model checking with net unfoldings. In: SPIN 2001. Volume 2057 of LNCS., Springer (2001) 37–56

9. Valmari, A.: On-the-fly verification with stubborn sets. In: Computer Aided Verification (CAV'93). Volume 697 of LNCS., Springer (1993) 397–408

10. Hansen, H., Penczek, W., Valmari, A.: Stuttering-insensitive automata for on-the-fly detection livelock properties. In: Formal Methods for Industrial Critical Systems. Volume 66(2) of Electronic Notes in Theoretical Computer Science., Elsevier (2002)

11. Donatelli, S.: Kronecker algebra and (stochastic) Petri nets: Is it worth the effort. In Colom, J.M., Koutny, M., eds.: Application and Theory of Petri Nets 2001. Volume 2075 of LNCS., Springer (2001) 1–18

12. Valmari, A.: Composition and abstraction. In Cassez, F., Jard, C., Rozoy, B., Ryan, M., eds.: Modelling and Verification of Parallel Processes. Volume 2067 of LNCS., Springer (2001) 58–99

13. Clarke, E., Long, D., McMillan, K.: Compositional model checking. In Parikh, R., ed.: Proc. 4th IEEE Symposium on Logic in Computer Science, IEEE Computer Society Press (1989) 353–362

14. Kurshan, R.: Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach. Princeton University Press (1994)

15. Vardi, M.Y., Wolper, P.: Automata-theoretic techniques for modal logic of programs. Journal of Computer and System Sciences **32** (1986) 183–221

16. Vardi, M.: An automata-theoretic approach to linear temporal logic. In: Logics for Concurrency: Structure versus Automata. Volume 1043 of LNCS. Springer (1996) 238–266

17. Clarke, E., Schlingloff, B.H.: Model checking. In Robinson, A., Voronkov, A., eds.: Handbook of Automated Reasoning. Elsevier (2001) 1637–1790

18. Thomas, W.: Automata on infinite objects. In Leeuwen, J., ed.: Handbook of Theoretical Computer Science. Volume B. Elsevier (1990) 133–191

19. Somenzio, F., Bloem, R.: Efficient Büchi automata from LTL formulae. In: Proceedings of the International Conference on Computer Aided Verification (CAV2000). Volume 1855 of LNCS., Springer (2000) 248–263

20. Gastin, P., Oddoux, D.: Fast LTL to Büchi automata translation. In: Computer Aided Verification (CAV'2001). Volume 2102 of LNCS., Springer (2001) 53–65

21. Giannakopoulou, D., Lerda, F.: From states to transitions: Improving translation of ltl formulae to büchi automata. In Peled, D., Vardi, M., eds.: Formal Techniques for Networked and Distributed Systems - FORTE 2002, Springer (2002) 308 – 326

22. Esparza, J., Heljanko, K.: A new unfolding approach to LTL model checking. Technical Report HUT-TCS-A60, Helsinki University of Technology (2000) Available from `http://www.tcs.hut.fi/Publications`.

23. Courcoubetis, C., Vardi, M., Wolper, P., Yannakakis, M.: Memory efficient algorithms for the verification of temporal properties. Formal Methods in System Design **1** (1992) 275–288

24. Latvala, T.: Model checking LTL properties of high-level Petri nets with fairness constraints. In: Applications and Theory of Petri Nets (ICAPTN'2001). Volume 2075 of LNCS., Springer (2001) 242–262

25. Varpaaniemi, K., Heljanko, K., Lilius, J.: PROD 3.2 — an advanced tool for efficient reachability analysis. In: Computer Aided Verification: 9th International Conference, CAV'97, Haifa, Israel, June 22–25, 1997, Proceedings. Volume 1254 of LNCS., Springer-Verlag (1997) 472–475
26. Petrucci, L.: Design and validation of a controller. In: Proceedings of the 4th World Multiconference on Systemics, Cybernetics and Informatics. Volume VIII., Orlando, FL, USA, International Institute of Informatics and Systemics (2000) 684–688
27. Dolev, D., Klawe, M., Rodeh, M.: An $O(n \log n)$ unidirectional distributed algorithm for extrema finding in a circle. Journal of Algorithms **3** (1982) 245–260