Model Checking Safety Properties in Modular High-Level Nets

Marko Mäkelä*

Helsinki University of Technology,
Laboratory for Theoretical Computer Science,
P.O.Box 9205, 02015 HUT, Finland
marko.makela@hut.fi
http://www.tcs.hut.fi/~msmakela/

Abstract. Model checking by exhaustive state space enumeration is one of the most developed analysis methods for distributed event systems. Its main problem—the size of the state spaces—has been addressed by various reduction methods.
Complex systems tend to consist of loosely connected modules, which may perform internal tasks in parallel. The possible interleavings of these parallel tasks easily leads to a large number of reachable global states. In modular state space analysis, the internal actions are explored separately in each module, and the global state space only includes synchronisations. This article introduces nested modular nets, which are hierarchal collections of nets synchronising via shared transitions, and presents a simple algorithm for model checking safety properties in modular systems.

Keywords: modular systems, state space enumeration, model checking, high-level nets

1 Introduction

Complex systems are often divided into modules that can be managed more easily. The internal structure of the modules is hidden behind high-level interfaces, the connection points for composing a complete system out of the components.

Abstracting from implementation details may make it easier to understand how a system works. However, these details may become significant when one wants to assert something about the behaviour of the composed system. To verify whether a desired property holds in the system, one could analyse all its reachable states. The question is whether the easily resulting *state space explosion* [18] can be ameliorated by utilising the division of the system into modules.

In a technique called *compositional reachability analysis* [19] or *modular verification* [7], a model is analysed in multiple phases. Sometimes, it is possible

* This research was supported by Jenny and Antti Wihuri Fund and by Academy of Finland (Project 47754).

W.M.P. van der Aalst and E. Best (Eds.): ICATPN 2003, LNCS 2679, pp. 201–220, 2003.

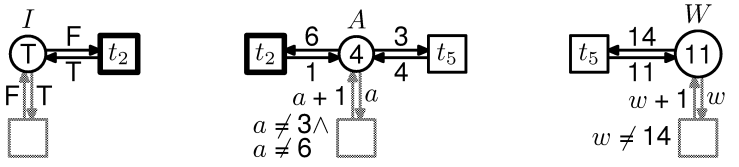


Fig. 1. A partial model of a control system. The gray rectangles denote internal transitions. The three nets synchronise on two labels, t_2 and t_5. The model can be flattened to a single net by fusing the transitions marked with t_2 and t_5.

to transform the property being checked into something that can be checked on each component separately or on a composition of some of the components. Also, the state space can be composed incrementally, collapsing the sequences of internal actions in each intermediate composition.

A modular state space exploration algorithm may save space and time compared to an algorithm for monolithic or flat models. Figure 1 corresponds to part of [16, Figure 1], which models a controller of automated guided vehicles. MARIA [15] constructed the reachability graph of the full model—30,965,760 nodes and 216,489,984 edges—in almost eight hours on a 1 GHz AMD Athlon system equipped with 1 GB of memory. In modular reachability analysis, the edges of the *synchronisation graph* are occurrences of synchronisations. In Figure 2, these are the black edges. Our algorithm constructs the synchronisation graph of the full model—512 nodes and 1,600 edges—in a split second.

Section 3 shows that it is safe to prohibit the occurrences of internal transitions in non-synchronising modules. This reduces the number of synchronisation states of the model in Figure 1 from six in Figure 2 to two.

Compositional or modular state space analysis has been presented for communicating state machines [7] and Petri nets. Input/output nets [10] communicate via dedicated places. Modular place/transition nets [5] use shared transitions. The techniques have also been sketched for high-level nets [1,4], but to our knowledge, no state space exploration algorithm has been presented before.

This paper describes an algorithm for checking safety properties in modular state spaces. Section 2 defines a class of modular high-level nets, and Section 3 defines state spaces for these nets. Section 4 describes our algorithm, and Section 5 reports experimental results. Finally, Section 6 concludes the presentation.

2 Nested Modular Nets

High-level nets are based on net graphs, consisting of the disjoint sets of places P and transitions T and the set of arcs $F \subseteq (P \times T) \cup (T \times P)$. Due to space constraints, we refer the reader to [2] for a definition of high-level nets.

Christensen and Petrucci define modular nets [4, Definition 2.1] as triples (S, PF, TF). The set S contains *modules*, which are high-level nets with no shared places or transitions. The sets $PF \subseteq 2^P$ and $TF \subseteq 2^T$ are the *place*

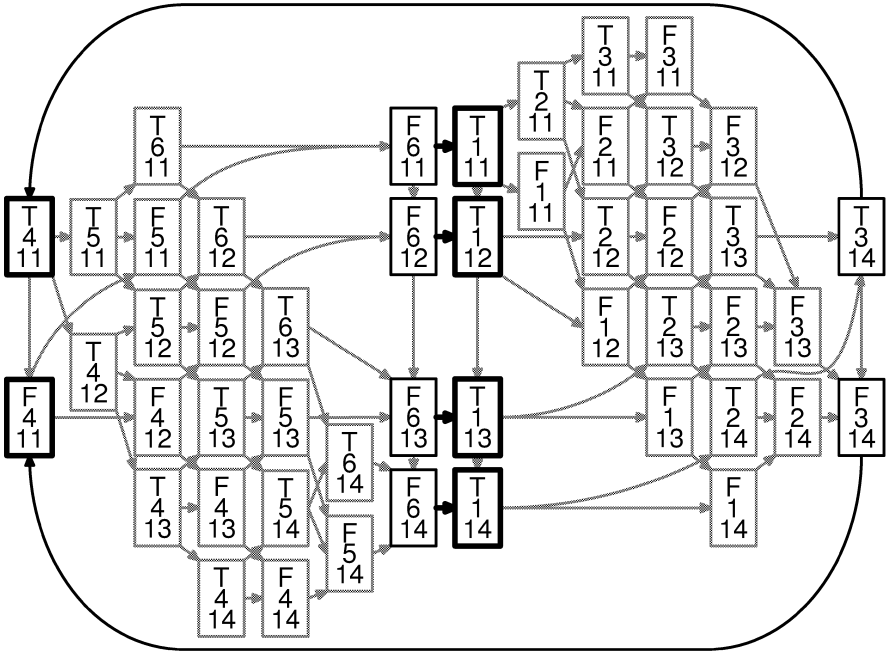


Fig. 2. The complete state space of the model presented in Figure 1, consisting of 48 reachable states and 98 transition occurrences. The initial state is the node at the top left. The gray edges and nodes denote the occurrences of internal transitions and the resulting states. The states immediately before or after a synchronisation are highlighted with a black border. There are four occurrences of t_2, the bold arrows in the centre of the picture, and two occurrences of t_5, the black edges leading from the right to the left of the figure.

fusion sets and *transition fusion* sets, respectively. The sets P and T refer to the combined sets of places and transitions in all modules.

The elements of the transition fusion set $tf \in TF$ are sets of transitions. They can be thought as *synchronisation labels*. If a transition of a module does not belong to any tf, it is an *internal transition*. Otherwise, it is an *external transition* that cannot occur on its own, but only in a synchronous step with other transitions in some tf to which it belongs.

Our definition of modular nets differs from [4, Section 2]. We make the simplifying assumption that there is no place fusion: $PF = \emptyset$. Since communication via shared places can be transformed into synchronisation via shared transitions [4, Section 5], it suffices to support the latter.

Furthermore, instead of defining one top-level structure containing basic nets as modules, we allow modules within modules. Cheung and Kramer motivate the use of subsystem hierarchies by analysing a model of a gas station [3, Section 2.4]. We believe that nested modules could be useful in the verification of multi-

layered protocols. Figure 3 illustrates how a modular model of a communication protocol can be reused as a module of a distributed application.

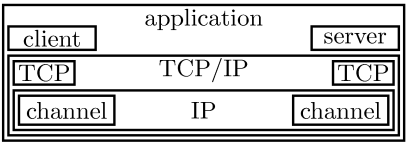


Fig. 3. A possible module hierarchy of a distributed Internet application. The TCP/IP layer consists of two TCP processes that synchronise with primitives provided by the IP layer, which encapsulates some channels. Client–server communication takes place via synchronisations with the TCP/IP module.

We shall define a nested modular net as a *hierarchy tree* of modules S.

Definition 1 (Hierarchy tree). *Let S be a finite set of high-level nets, such that each $s \in S$ contains sets of places P_s and transitions T_s, and $(P_{s_1} \cup T_{s_1}) \cap (P_{s_2} \cup T_{s_2}) = \emptyset$ for all $s_1, s_2 \in S, s_1 \neq s_2$. A* hierarchy tree H of S *is a connected directed graph $H = (S, C \subset S \times S)$ for which the following hold:*

1. *H has a unique* root *$s_0 \in S$, such that $C(s) \neq s_0$ for all $s \in S$, and*
2. *other nets in H have a unique ancestor: for each $s \in S \setminus \{s_0\}$, there exists exactly one $s' \in S$ such that $s \in C(s')$.*

A modular net (S, PF, TF) of Christensen and Petrucci can be represented as a nested modular net that has a root module $s_0 \notin S$ with $P_{s_0} = T_{s_0} = \emptyset$ and a hierarchy tree $(\{s_0\} \cup S, \{s_0\} \times S)$.

The fusion sets TF and PF shall be restricted in such a way that only modules with a common parent in the tree can synchronise with each other.

Definition 2 (Hierarchal fusion sets). *Let $H = (S, C)$ be a hierarchy tree, and let P_s and T_s be as in Definition 1. Let $T = \bigcup_{s \in S} T_s$ and $P = \bigcup_{s \in S} P_s$. The sets $TF \subseteq 2^T$ and $PF \subseteq 2^P$ are* fusion sets*.*

TF (or PF) is H-hierarchal *if for all $tf \in TF$ (or $pf \in PF$), the modules of the transitions included in tf (or places in pf) are siblings: for all $(s_1, s_1'), (s_2, s_2') \in C$, either $s_1 = s_2$ or $T_{s_1'} \cap tf = \emptyset \vee T_{s_2'} \cap tf = \emptyset$ (or $P_{s_1'} \cap pf = \emptyset \vee P_{s_2'} \cap pf = \emptyset$).*

A transition $t \in T_s$ of a module $s \in S$ is an *internal transition* if there is no $tf \in TF$ such that $t \in tf$. If $t \in tf$ for some $tf \in TF$, t *synchronises on* tf.

The state of a collection of nets is an assignment of markings for each net. In our hierarchy tree, we can define the state of a subtree of nets as an assignment of markings for the root of the subtree and for all its descendants in the tree. These are included in the transitive closure of the child relation:

Definition 3 (Transitive closure). *Let $C \subseteq S \times S$ be a binary relation. Then $C^+ \subseteq S \times S$ is the* transitive closure of C*, the smallest relation that fulfils the following definition.*

1. $(s, s') \in C \Rightarrow (s, s') \in C^+$,
2. $(s, s') \in C^+ \wedge (s', s'') \in C^+ \Rightarrow (s, s'') \in C^+$.

Next, we shall define a transformation that imports the places of the descendent nets into each ancestor net in the hierarchy tree. For each transition fusion set $tf \in TF$, the transformation also instantiates a *synchronisation transition* by the same name tf. A transition $t \in tf$ is enabled if and only if all transitions $t' \in tf$ are enabled—or tf is enabled in the parent.

Definition 4 (Nested modular net). *Let $H = (S, C)$ be a hierarchy tree, and let TF be a H-hierarchal transition fusion set, and let $PF = \emptyset$ be the place fusion set of S. Let $s \in S$ be a module with the sets of places P_s, transitions T_s and arcs F_s. The* modular augmentation $\mathcal{M}(s)$ of s *is defined as $\mathcal{M}(P_s) = P_s \cup P'_s$, $\mathcal{M}(T_s) = T_s \cup T'_s$ and $\mathcal{M}(F_s) = F_s \cup F'_s$ as follows.*

1. *a) $P'_s = \bigcup_{s' \in C^+(s)} P_{s'}$ (import all places from the descendants)*
   b) $T'_s = \{tf \in TF : \exists s' \in C(s) : T_{s'} \cap tf \neq \emptyset\}$ (transform the fusions between transitions in child nets into synchronisation transitions in the parent)
2. *each synchronisation transition $tf \in T'_s$ is a fusion of the transitions $t \in tf$:*
   a) the set of variables of tf is the union of the sets of variables of $t \in tf$.
   b) the guard of tf is the conjunction of the guards of $t \in tf$.
   c) for each $t \in tf$, if there is an arc $f = (t, p)$ or $f = (p, t)$ such that $t \in T_{s'}$ and $f \in F_{s'}$ for some $s' \in C(s)$, then there is an arc $f' = (tf, p)$ or $f' = (p, tf)$ in F'_s, respectively. The label of f' is that of f.

The triple $(H, TF, \mathcal{M})$ is a nested modular net.

Definition 5 (Markings and projected markings). *Let $(H, TF, \mathcal{M})$ be a nested modular net with $H = (S, C)$. Let $s \in S$, and let M be a marking of $\mathcal{M}(s)$. For $s' \in C^+(s)$, the* projection of M on $\mathcal{M}(s')$ *is $M_{s'} := \bigcup_{p \in \mathcal{M}(P_{s'})} \{(p, M(p))\}$ where s' has the set of places $P_{s'}$.*

Definition 6 (Occurrence rule for nested modular nets). *Let $(H, TF, \mathcal{M})$ be a nested modular net and let $H = (S, C)$. Let $s \in S$ be a net with the transition set T_s. Let $t \in \mathcal{M}(T_s)$ and let m be an assignment for the variables of t. In a given marking M_1 of $\mathcal{M}(s)$,* transition t is $\mathcal{M}(s)$-enabled in mode m *if*

1. $t \in T_s$**:** $M^* = M_1$
   $t \notin T_s$**:** $M^* = M_1$ *or there is a sequence of internal transitions $t_1 \ldots t_n$ such that $M_1 \xrightarrow{t_1} M_2 \cdots \xrightarrow{t_n} M^*$ and each t_i belongs to some $s' \in C(s)$ that synchronises on t—i.e., each s' has a transition $t' \in T_{s'}$ such that $t' \in t$,*
2. *t is enabled in mode m in M^*.*

Then, t may $\mathcal{M}(s)$-occur in mode m, which results in the successor marking M' *that is obtained by firing t in mode m in the marking M^*.*

Definition 6 modifies the occurrence rule of the underlying high-level nets by defining special treatment of synchronisation transitions. By Definition 4, a transition corresponding to a synchronisation label $t \notin T_s, t \in TF$ is a fusion of the

transitions of the child nets $s' \in C(s)$ that synchronise on the label, or belong to the set t. The fused transition t can only be enabled if all its components in the child nets are enabled in some marking M^* reachable from M_1 via a possibly empty sequence of internal transitions.

The synchronisation graph of a modular net can be constructed by exploring its $\mathcal{M}(s_0)$-enabled transitions, where s_0 is the root of the hierarchy tree. This graph only contains the occurrences of synchronisation transitions.

The reachability graph of a modular net can be computed by flattening the hierarchy to an ordinary high-level net and applying the occurrence rule of ordinary high-level nets on the flattened net. The reachability graph may contain more occurrences of synchronisation transitions than the synchronisation graph, since the occurrences of the internal transitions of child nets are not restricted.

Definition 7 (Flattened nested modular net). *Let $(H, TF, \mathcal{M})$ be a nested modular net with the hierarchy tree $H = (S, C)$ whose root is s_0. For each $s \in S$, let there be the set of places P_s, the set of transitions T_s and the set of arcs $F_s \subseteq (P_s \times T_s) \cup (T_s \times P_s)$. Let $P = \bigcup_{s \in S} P_s$ and $T = \bigcup_{s \in S} T_s$. Then $\mathcal{F}(H, TF, \mathcal{M})$ is the* flattened nested modular net of $(H, TF, \mathcal{M})$*, with the following elements:*

1. $P = \mathcal{M}(P_{s_0}) = \bigcup_{s \in S} P_s$ *(all the places of all nets),*
2. $T = \bigcup_{s \in S} \mathcal{M}(T_s) \setminus \bigcup_{tf \in TF} tf$ *(all internal transitions),*
3. $F = \bigcup_{s \in S} \mathcal{M}(F_s) \cap ((P \times T) \cup (T \times P))$ *(the arcs attached to the transitions).*

Definition 7 also imports synchronisation transitions $tf \in TF$ to the flattened net, unless they are external, i.e., $tf \in tf' \in TF$. Only the "outermost" synchronisation transitions tf' (such that $tf' \notin tf''$ for all $tf'' \in TF$) are imported.

3 Modular State Spaces

Next, we shall define the reachability graph of a high-level net and the synchronisation graph of a nested modular high-level net. Both are called *state spaces*. We shall also define the *equivalent state space* of a nested modular net and prove that it is equivalent to the state space of a flattened nested modular net.

Definition 8 (State space). *Let s be a high-level net with the initial marking M_0. The* state space of s *is a directed rooted graph $G = (V, E, v_0)$, with $E \subseteq V \times V$ and $v_0 \in V$, the smallest graph for which the following hold:*

1. *$v_0 = M_0$ is the* initial state*,*
2. *for each $M \in V$, if $M \longrightarrow M'$, then $M' \in V$ and $(M, M') \in E$.*

Let $(H, TF, \mathcal{M})$ be a nested modular net with the root net s_0. The state space of $(H, TF, \mathcal{M})$ *is defined analogously, with v_0 corresponding to the initial marking of $\mathcal{M}(s_0)$ and with the edges in E corresponding to the occurrences of $\mathcal{M}(s_0)$-enabled transitions.*

The three nets s_1, s_2, s_3 in Figure 1 can be interpreted as $(H, TF, \mathcal{M})$, such that $H = (S, C)$, $S = \{s_0, s_1, s_2, s_3\}$, s_0 is empty, and $C = \{s_0\} \times \{s_1, s_2, s_3\}$. The synchronisations are $TF = \{\{t_2^1, t_2^2\}, \{t_5^2, t_5^3\}\}$, where the superscripts identify the

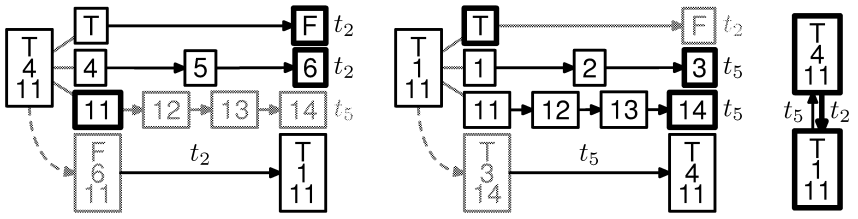


Fig. 4. The construction of the synchronisation graph of the controller model (Figure 1). In the initial marking, only t_2 is enabled, as all modules that synchronise on it can internally reach a state where it is enabled. Similarly, only t_5 is enabled in the successor state. In a synchronisation, the states of non-participating modules do not change.

modules, e.g., $t_2^2, t_5^2 \in T_{s_2}$. Figure 2 represents the state space of $\mathcal{F}(H, TF, \mathcal{M})$. The state spaces of $(H, TF, \mathcal{M})$ are illustrated in Figure 4. We assert that the two synchronisation states correspond to the two groups of states enclosed in a thick border in Figure 2. These groups can be formalised as follows.

Definition 9 (Related markings [4, Definition 3.3.1]). *Let $(H, TF, \mathcal{M})$ be a nested modular net. Let $H = (S, C)$ and $s \in S$. Let $\mathbb{M}_s$ be the set of all markings of $\mathcal{M}(s)$. Let Π_s map each marking $M \in \mathbb{M}_s$ to a set of markings reachable via $\mathcal{M}(s')$-enabled internal transitions of the modules $s' \in C(s)$. Let $R_s \subseteq \mathbb{M}_s \times \mathbb{M}_s$ identify markings with common internal successor states: $(M_1, M_2) \in R_s \Leftrightarrow \Pi_s(M_1) \cap \Pi_s(M_2) \neq \emptyset$. Let R_s^+ be the transitive closure of R_s.*

Similar to Christensen and Petrucci [4], checking whether a state M' is in $\Pi_s(M)$ does not require $\Pi_s(M)$ to be generated—it is sufficient to check that in each module $s' \in C(s)$, the local component $M'_{s'}$ is either reachable from the local component $M_{s'}$ via internal transitions, or $M'_{s'} \in \Pi_{s'}(M_{s'})$.

The left half of Figure 2 represents the initial state of the model in Figure 1—$M_0 = \{(I, \mathsf{T}), (A, 4), (W, 11)\}$—and the internal states $\Pi_{s_0}(M_0)$. Let $M_1 = \{(I, \mathsf{F}), (A, 4), (W, 11)\}$. Clearly, $\Pi_{s_0}(M_0) \cap \Pi_{s_0}(M_1) \neq \emptyset$, and thus R_{s_0} contains (M_0, M_1). Note that by definition, R_{s_0} is reflexive and symmetric. $R_{s_0}^+$ is also transitive and thus an equivalence relation. In this example, $R_{s_0}^+ = R_{s_0}$.

Next, we shall define a mapping for the state space of a nested modular net and prove that it equals the state space of the corresponding flattened net.

Definition 10 (Equivalent state space [4, Definition 3.3.3]). *Let $G = (V, E, v_0)$ be the state space of the nested modular net $(H, TF, \mathcal{M})$ with $H = (S, C)$. Let s_0 be the root of H, and let $\mathbb{M}$ and R^+ be as in Definition 9. The* equivalent state space of G *is $G' = (V', E', v_0)$, defined inductively as follows:*

1. $V' = \bigcup_{M \in V} R_{s_0}^+(M)$
2. *for all states $v, v' \in V'$, $(v, v') \in E$ if $v \xrightarrow{t,m} v'$ where*
   a) *t is $\mathcal{M}(s_0)$-enabled in mode m in the marking v, or*
   b) *t is an internal transition of some $s \in C(s_0)$, and t is $\mathcal{M}(s)$-enabled in mode m in the marking v.*

Proposition 1 (Equivalence of state spaces [4, Theorem 3.3.4]). *Let $(H, TF, \mathcal{M})$ be a nested modular net and $\mathcal{F}(H, TF, \mathcal{M})$ be a flattened nested modular net. Let $G_{\mathcal{F}} = (V_{\mathcal{F}}, E_{\mathcal{F}}, v_0)$ be the state space of $\mathcal{F}(H, TF, \mathcal{M})$. Let $G = (V, E, v_0)$ be the state space of $(H, TF, \mathcal{M})$, and let $G' = (V', E', v_0)$ be the equivalent state space of G. Then $G_{\mathcal{F}} = G'$.*

By Definitions 7 and 10, each state space has the same initial state v_0. The sets of potential markings coincide for the modular and the flattened net, as they both contain the same set of places, by Definition 7.

Compared to the flattened net (Definition 7), the occurrence rule for nested modular nets (Definition 6) hides the occurrences of internal transitions. Thus, for each $(M, M') \in E$, there is a path $(M, M_1), (M_1, M_2), \ldots, (M_n, M') \in E_{\mathcal{F}}$ where the intermediate states $M_1 \ldots M_n$ result from the occurrences of internal transitions. Thus, $V \subseteq V_{\mathcal{F}}$ and $E \subseteq E_{\mathcal{F}}^+$. By Definitions 9 and 10, V' and E' are extended from V and E by adding states and edges corresponding to the occurrences of internal transitions. Since $G_{\mathcal{F}}$ and G' only differ by these occurrences, we have $V_{\mathcal{F}} = V'$ and $E_{\mathcal{F}} = E'$.

4 Checking Safety Properties

4.1 Algorithm for Exhaustive Modular State Space Exploration

Figure 5 presents a basic algorithm for checking safety properties by exhaustive modular state space enumeration. If we ignore the right column of this figure and the invocation of MODULES in the procedure EXPLORE, we have the basic exploration algorithm for flat state spaces.

The procedure EXPLORE constructs the state space of a net $\mathcal{M}(s)$ by exploring all enabled transitions in each state reachable from M_1. It invokes TRANSITIONS in each state $M \in V$. For each enabled transition of s, TRANSITIONS invokes REPORT in order to insert previously unknown states $M' \notin V$ into the search queue Q and into the set of reachable states V.

While the procedure TRANSITIONS computes the successor states of M by exploring the enabled transitions of s, the procedure MODULES explores the synchronising transitions of each module $s' \in C(s)$ and invokes SYNC to compute the synchronisations. The transition enabling test invoked by TRANSITIONS and SYNC has been described in [13].

By invoking EXPLORE on each s', the procedure MODULES constructs a synchronisation relation $\mathcal{S}$ that maps child nets to synchronisation labels and local states where these synchronisations are enabled. If $(s', tf, M^{s'}) \in \mathcal{S}$, the marking $M^{s'}$ of s' is reachable from $M_{s'}$—the projection of M on $\mathcal{M}(s')$—via internal transitions, and a transition synchronising on tf is enabled in $M^{s'}$. Each invocation of MODULES is associated with such a relation $\mathcal{S}$. The relation is extended by TRANSITIONS and explored by SYNC.

The procedure SYNC iterates over certain subsets of the synchronisation relation $\mathcal{S}$. MARIA implements the relation with some mappings and arrays. The iteration has been implemented as a recursive loop over those modules $s' \in C(s)$

Explore s starting from M_1

EXPLORE$(s, M_1, \mathcal{S})$:
$V := \{M_1\}; E := \emptyset; Q := (M_1)$
while $Q \neq ()$
  $M := Q$.remove()
  TRANSITIONS$(s, M, V, E, Q, \mathcal{S})$
  MODULES(s, M, V, E, Q)

Report the successors of M in s

TRANSITIONS$(s, M, V, E, Q, \mathcal{S})$:
for each t, m, M' **s.t.** $M \xrightarrow{t,m} M'$ **in** s:
  for each $tf \in TF$ **s.t.** $t \in tf$:
    $\mathcal{S} := \mathcal{S} \cup \{(s, tf, M)\}$
  if not exists $tf \in TF$ **s.t.** $t \in tf$
    REPORT(M, M', V, E, Q)

REPORT(M, M', V, E, Q):
$E := E \cup \{(M, M')\}$
if $M' \notin V$
  $V := V \cup \{M'\}$
  if error(M') **show trace to** M'
  else Q.insert(M')

Explore the modules of s from M

MODULES(s, M, V, E, Q):
$\mathcal{S} := \emptyset$
for each $s' \in C(s)$:
  EXPLORE$(s', M_{s'}, \mathcal{S})$
for each $tf \in \mathcal{M}(T_s) \setminus T_s$:
  SYNC$(s, M, V, E, Q, \mathcal{S}, tf)$

Compute the synchronisations on tf

SYNC$(s, M, V, E, Q, \mathcal{S}, tf)$:
for each $\bigcup_{\substack{s' \in C(s) \\ tf \cap T_{s'} \neq \emptyset}} \{(s', tf, M^{s'})\} \subseteq \mathcal{S}$:
  $M^* := M; \forall (p, q) \in \bigcup_{\substack{s' \in C(s) \\ tf \cap T_{s'} \neq \emptyset}} M^{s'}$: $M^*(p) := q$
  for each tf, m, M' **s.t.** $M^* \xrightarrow{tf,m} M'$ **in** $\mathcal{M}(s)$:
    REPORT(M, M', V, E, Q)

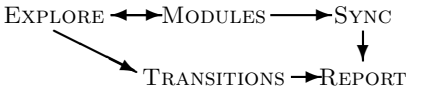


Fig. 5. The basic algorithm for modular state space analysis and its call graph. All parameters are passed by reference. The synchronisation information $\mathcal{S}$ is associated with each invocation of MODULES. V is a set of encountered states, E is a set of edges and Q is a queue of unexplored states. TRANSITIONS assumes that there is at most one $t \in T_s \cap tf$. If $t_1, \ldots, t_n \in T_s \cap tf$, the transitions $t_1, \ldots, t_n$ will have to be fused together in a pre-processing step. The algorithm is invoked for a nested modular net $(H = (S, C), TF, \mathcal{M})$ as EXPLORE$(s_0, M_0, \mathcal{S})$ where s_0 is the root net, M_0 is the initial marking of $\mathcal{M}(s_0)$ and $\mathcal{S} = \emptyset$.

that synchronise on tf. On each round, a marking $M^{s'}$ of s' is assigned to M^*. That is, M^* is first initialised to M, and the markings of the synchronising modules $\mathcal{M}(s')$ will be substituted in M^*.

An Example Run. We shall demonstrate the algorithm with the model presented in Figure 1 and discussed near Definition 9. The root net s_0 of the model contains no places or transitions. The modular augmentation $\mathcal{M}(s_0)$ is depicted in Figure 6. Its initial marking is $M_0 = \{(I, \mathsf{T}), (A, 4), (W, 11)\}$.

The algorithm is invoked as EXPLORE$(s_0, M_0, \emptyset)$. It initialises the set V and the queue Q with the initial marking M_0. It removes the marking from the queue and passes it as a parameter to TRANSITIONS. Since s_0 has no transitions (but $\mathcal{M}(s_0)$ has), invoking TRANSITIONS does not change anything.

Next, EXPLORE invokes the procedure MODULES. The left part of Figure 4 shows how MODULES splits the marking into the markings of child nets and invokes EXPLORE on each of them. Let us look at the EXPLORE call for s_1, the leftmost net in Figure 1. On the first round of the **while** loop, TRANSITIONS calls

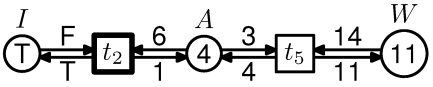


Fig. 6. The modular augmentation $\mathcal{M}(s_0)$ of the empty root net s_0 of the control system presented in Figure 1.

REPORT$(\{(I,\mathsf{T})\},\{(I,\mathsf{F})\},V,E,Q)$, which inserts $\{(I,\mathsf{F})\}$ into the local search queue of EXPLORE$(s_1,\{(I,\mathsf{T})\},\mathcal{S})$. Thus, the **while** loop of EXPLORE enters a second round, with $M=\{(I,\mathsf{F})\}$. For this marking, TRANSITIONS finds out that the transition synchronising on t_2 is enabled. Thus, it adds $(s_1,t_2^1,\{(I,\mathsf{F})\})$ to the synchronisation set $\mathcal{S}$. After this, EXPLORE returns to MODULES.

By invoking EXPLORE on all nets $C(s_0)=\{s_1,s_2,s_3\}$, MODULES constructs the set $\mathcal{S}=\{(s_1,t_2^1,\{(I,\mathsf{F})\}),(s_2,t_2^2,\{(A,\mathsf{6})\}),(s_3,t_5^3,\{(W,\mathsf{14})\})\}$. Next, it invokes SYNC on t_2 and t_5. For t_2, SYNC finds exactly one subset of $\mathcal{S}$, namely $\{(s_1,t_2^1,\{(I,\mathsf{F})\}),(s_2,t_2^2,\{(A,\mathsf{6})\})\}$. The markings $M=M_0$, $M^{s_1}=\{(I,\mathsf{F})\}$ and $M^{s_2}=\{(A,\mathsf{6})\}$ are combined to $M^*=\{(I,\mathsf{F}),(A,\mathsf{6}),(W,\mathsf{11})\}$. In M^*, the transition t_2 is enabled, and a successor marking $\{(I,\mathsf{T}),(A,\mathsf{1}),(W,\mathsf{11})\}$ of M is reported. For t_5, there is no subset $\{(s_2,t_5^2,M^{s_2}),(s_3,t_5^3,M^{s_3})\}$ of $\mathcal{S}$, since there is no $(s_2,t_5^2,M^{s_2})\in\mathcal{S}$. This can be compared to the left part of Figure 4.

So, MODULES returns to EXPLORE after having recorded one successor marking. The second and last round of the **while** loop in EXPLORE$(s_0,M_0,\emptyset)$ proceeds in a similar fashion; see the middle part of Figure 4.

It should be noted that the parameter M of REPORT and the sets E do not affect the control flow of the algorithm, and thus they need not be implemented.

Correctness. We assert that the algorithm presented in Figure 5 computes state spaces (Definition 8) of nested modular nets.

Definition 6, the occurrence rule for nested modular nets, has one essential addition to the occurrence rule of the underlying nets. The case $t\notin T_s$ deals with synchronisations, preceded by a sequence of internal transitions. Similarly, the algorithm in Figure 5 extends the basic state space exploration algorithm [14, Algorithm 1] with the procedures MODULES and SYNC that explore the internal transitions of modules and make all possible synchronisations occur.

The subroutine EXPLORE terminates when all states have been explored, or the search queue Q runs out. States are added to Q by REPORT, which is the only routine altering Q, the set of reachable states V and the set of edges E.

When EXPLORE is invoked on a net s that contains no module, $C(s)=\emptyset$, the invocation of MODULES in the **while** loop of EXPLORE does not affect anything. Clearly, each invocation of TRANSITIONS augments V and E with the successor states of M—those states that result from the occurrences of internal transitions. Since each reachable state is inserted into Q exactly once and since EXPLORE invokes TRANSITIONS on each state in Q, it is easy to see that after the **while** loop terminates, the sets V and E correspond to the state space of s, with

$v_0 = M$. Provided that at most one transition of s synchronises on any given tf (see the caption of Figure 5), the set $\mathcal{S}$ will contain a tuple for each enabled external transition of the net and for each reachable state where it is enabled.

The procedure MODULES becomes significant when a net s contains modules. It explores the reachable states in each module $s' \in C(s)$, starting from $M_{s'}$, the current marking M projected on s'. Once all invocations of EXPLORE have returned to MODULES, the set $\mathcal{S}$ contains an item for each module and for each state where a synchronisation is possible. The outermost **for each** loop in SYNC iterates over all possible synchronisation points on tf and initialises a marking M^* on each iteration. The markings M^* are reachable from M by the occurrences of internal transitions of those modules that synchronise on tf. This is equivalent to the case $t \notin T_s$ of Definition 6. Finally, SYNC generates the successor states of M by making each enabled instance of tf occur in each M^*.

We conclude that EXPLORE$(s_0, M_0, \emptyset)$ constructs the state space of a modular algebraic system net $(H, TF, \mathcal{M})$ with the root s_0 and the initial state M_0.

4.2 Specifying Safety Properties

The procedure REPORT in Figure 5 checks a safety property on a newly generated state. Erroneous states are not explored further—they are reported to the user.

The safety model checker in MARIA recognises three kinds of erroneous states:

- states that satisfy a "reject" or "deadlock" formula,
- states that cannot be compacted due to a constraint violation, and
- states whose successors cannot be computed due to an evaluation error.

A "reject" formula is a Boolean condition on reachable markings. A "deadlock" formula is a condition on reachable markings where no transition is enabled.

More generic properties can be specified in linear temporal logic. It covers infinite executions, but its "safety" subset [12] is equivalent to finite state automata, which deal with finite execution sequences. The property "whenever A becomes 2, it will remain less than 5" does not hold in the net s_2 of Figure 1, since its place A acts a counter from 1 to 6. In the automata-theoretic approach to verification, a desired property of a system is negated and translated into an automaton. The system is in error if an accepting state (the dashed one in Figure 7) is reachable in the product automaton of the system's state space and the automaton corresponding to the negated property.

We would like to synchronise a modular state space with a property automaton. In the left part of Figure 4, $A \neq 2$ and the property automaton of Figure 7 remains in its initial state 0. In the middle part, the markings of A are 1, 2 and 3. In the state $A = 2$, the automaton moves to the state 1, but it cannot move to its accepting state 2, even though the system clearly violates the property!

Obviously, the property automaton must be fused with the relevant module, as in Figure 8. Figure 9 shows the state space of the fused model and the local states that are reachable from the last synchronisation. The property is violated in the dashed state. We see that the product of a property and a model may have more states than the plain model (Figure 4). Thus, it may be wise to check this kind of properties one at a time to avoid a combinatorial explosion.

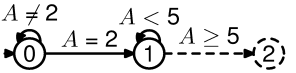


Fig. 7. A finite automaton for the formula $\neg\Box\,((A = 2) \Rightarrow \Box(A < 5))$.

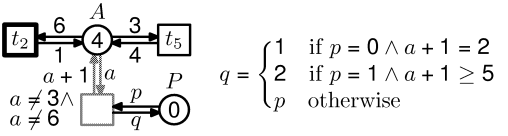


Fig. 8. The automaton of Figure 7 composed with the net s_2 of Figure 1. Only the occurrences of the internal transition can change the state of the automaton.

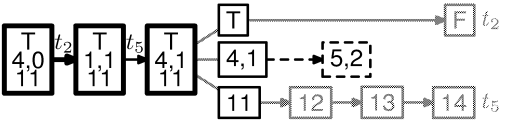


Fig. 9. An error trace of the property (Figure 7) and the model (Figures 1 and 8).

4.3 Constructing Error Traces

An *error trace* is a sequence of model actions that leads from the initial state to an erroneous state or transition. The algorithm presented in Figure 5 does not construct error traces—it is only aware of the last state on the trace.

The information needed for producing error traces should be stored in as little space as possible, so that more memory is available for accommodating the sets of encountered states and the collections of unprocessed states. Some information can be omitted and recomputed when an error is detected. This allows the verification of larger systems and more complex properties. For instance, the complete set of edges E may need much more storage space than the set of reachable states V if the state space contains many cycles and branches, or the edges are labelled with the names and firing modes of the occurring transitions.

Efficient production of an error trace requires a function $\mathsf{ancestor} : \mathbb{M} \to \mathbb{M}$ that maps each new state to the state from which it was obtained (the parameters M and M' of REPORT in Figure 5, $M' \notin V$). All states on an error trace can be enumerated by repetitively applying this function on the error state until the first state M_1 is reached. Once all the states $M_1, \ldots, M_n$ of the trace are known, the transitions can be obtained by computing the successor states of each state M_i in the trace and displaying a transition leading from M_i to M_{i+1}. There might not be a unique shortest error trace—this method produces one of them.

The function ancestor could be defined as something that follows the edge relation E backwards. Alas, we cannot store E, as we want to preserve memory. Stern and Dill [17] propose an addition to the algorithm: whenever a state

M' is inserted into Q, it is also appended to an auxiliary file together with the position of its ancestor M in the said file. The collection Q must associate each unprocessed state with these file positions. This file provides the mapping ancestor.

Writing the counterexample recovery information to a file does not significantly affect the performance, since sequential file access is fast. Only when a counterexample trace is produced, random (slow) access is needed. Even at that point, the input/output overhead may be insignificant.

Our implementation of MODULES (Figure 5) shows an error trace to M if any of the EXPLORE invocations reports an error. The tool can be told to stop after reporting a specified number of errors. When the model in Figure 1 is checked for the property in Figure 7, MARIA reports the trace (part of Figure 9) in two parts: from $\{(A, \mathsf{4}), (P, \mathsf{1})\}$ to $\{(A, \mathsf{5}), (P, \mathsf{2})\}$, and from the initial state of the modular system to the synchronisation state $\{(I, \mathsf{T}), (A, \mathsf{4}), (P, \mathsf{1}), (W, \mathsf{11})\}$.

This arrangement produces short error traces—in fact, the produced traces are as short as possible if the state spaces are constructed in breadth-first order.

5 Experiments

5.1 Automated Guided Vehicles

The first system [16, Figure 1] that was analysed with our algorithm models the coordination of automated guided vehicles on a factory floor. The state space of the flattened model consists of 30,965,760 nodes and 216,489,984 edges. The model is distributed with MARIA in the file `modular.pn`.

For the modular model, the algorithm in Figure 5 produces a state space of 836 nodes and 2,644 edges. This state space consists of 325 strongly connected components, one of which is terminal. This is somewhat surprising, since the state space of the flattened model consists of a single strongly connected component. Each of the remaining 324 components consists of a single state. Thus, there are no cycles between the 324 states—all edges lead towards the terminal component.

In the initial marking in [16, Figure 1], several modules are in an *intermediate state* in the sense that some internal transitions have occurred after synchronisations. Starting from a marking where the occurrences of these internal transitions have been undone, we obtained 512 nodes and 1,600 edges. This corresponds to the terminal strongly connected component of the original state space.

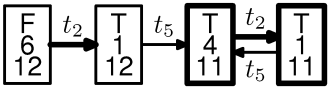
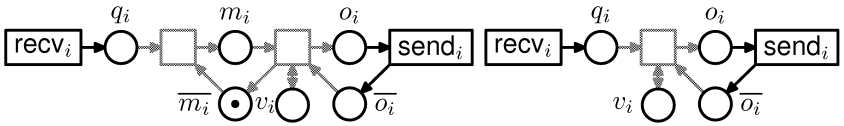


Fig. 10. The state space of the modular model (Figure 1) with an initial marking corresponding to [16, Figure 1]. The state space in Figure 4 is smaller.



(a) with intermediate buffer m_i (b) direct consumption from q_i

Fig. 11. A schematic view of two alternative implementations of the modules s_i of the leader election protocol, $1 \leq i \leq n$. The initial markings of v_i and o_i and the arc inscriptions have been omitted, and q_i should be accessed as a queue. The composition is $(H, TF, \mathcal{M})$ with $H = (S, C)$, $S = \{s_0, \ldots, s_n\}$ (s_0 is empty), $C = \{s_0\} \times \{s_1, \ldots, s_n\}$ and $TF = \bigcup_{i=1}^{n} \{\{\mathsf{send}_i, \mathsf{recv}_{(i \bmod n)+1}\}\}$.

Table 1. State space sizes and exploration times on a 1.67 GHz AMD Athlon XP for the leader election protocol (Figure 11) for different numbers of processes.

	flat (a)			flat (b)			modular		
n	nodes	edges	time	nodes	edges	time	nodes	edges	time
3	155	299	0.0 s	69	126	0.0 s	33	63	0.0 s
4	712	1,847	0.0 s	240	588	0.0 s	90	227	0.0 s
5	3,428	11,194	0.1 s	870	2,693	0.0 s	251	800	0.0 s
6	16,788	66,039	0.8 s	3,213	12,013	0.1 s	713	2,746	0.1 s
7	82,663	380,263	5.3 s	11,949	52,310	0.6 s	2,041	9,210	0.2 s
8	407,695	2,146,961	31.6 s	44,544	223,338	2.9 s	5,863	24,267	0.9 s

In Figure 1, if the internal transition in the middle occurs twice and the ones at the sides occur once, the result corresponds to a subset of [16, Figure 1]. Figure 10 shows the state space starting from this marking.

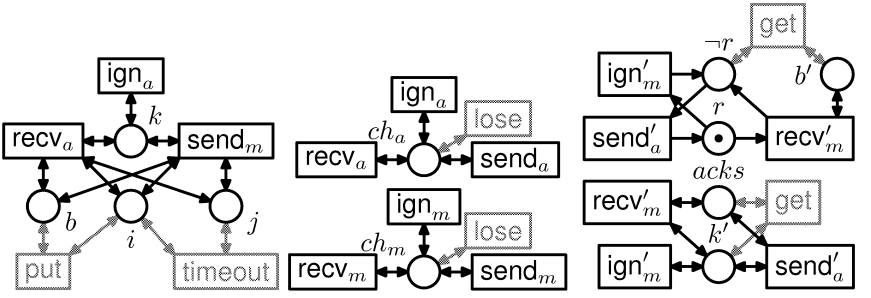
5.2 Leader Election in Unidirectional Ring

One of the examples distributed with Spin [8] is the leader election protocol in a unidirectional ring [6]. Figure 11 depicts the operation of the modules. Each module s_i has an input queue q_i, from which it takes messages m_i that are processed, affecting the local variables v_i. The modules also contain an output buffer o_i that can hold at most one message. In the initial state, the output buffers o_i are filled with the initiating messages of the protocol.

According to Table 1, the modules sketched in Figure 11(a) generate much bigger state spaces for the flattened net than those shown in Figure 11(b). The state space of the root net is the same for both variations.

Karaçalı and Tai [11] have modelled the system with extended finite state machines. The flat state spaces reported in [11, Table 1] are an order of magnitude bigger than those in Table 1. However, their reduction algorithm appears to outperform modular analysis by generating only $8n + 13$ states and $8n + 12$ events for systems consisting of $3 \leq n \leq 6$ processes.

The algorithm of Karaçalı and Tai [11, Section 5] makes use of a transition dependency relation [11, Section 4]. Such relations are difficult to derive for high-



(a) the producer–sender (b) the lossy channels (c) the recipient–consumer

Fig. 12. The modules of the sliding window protocol. The inscriptions have been omitted. The message and acknowledgement channels support the actions send (send a message wr), recv (receive a message rd) and ign (ignore rd). The parameters rd and wr are shared variables among the synchronising transitions.

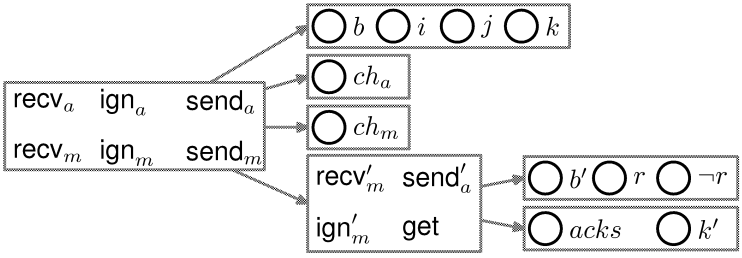


Fig. 13. The hierarchy tree of the sliding window protocol model (Figure 12). The modules s are denoted by gray rectangles, and gray arrows represent the child relation C. The synchronisation labels and the places are indicated for each module. The recipient–consumer module defines four synchronisation labels, three of which synchronise on the similarly named labels of the root module.

level Petri nets, which do not have separate control and data flows. Karaçalı and Tai have improved partial order reduction with something that resembles modular state space exploration. It would be interesting to see if partial order reduction could be efficiently implemented on top of modular analysis.

5.3 Sliding Window Protocol

Figures 12 and 13 show our modular model of the sliding window protocol, which is distributed with Maria in the file `swn-m.pn`. There are two communication channels, which are connected to a sending and a receiving process. The producer entity that feeds the sending process is modelled by the internal transition put, and the consumer is modelled by the transitions get. If the channels lose messages, a sender timeout will eventually occur.

Both channel modules in Figure 12 contain one place, which holds one token that represents the contents of the queue. The producer–sender has a transmission buffer b for tw items and three indices to it: i and k to the bounds of the transmission window and j to the next item to be sent ($i \leq j \leq k$). The receiver has a buffer b' for rw items, the index number k' of the next awaited message and an array *acks* that indicates which messages have been acknowledged.

The recipient–consumer module is divided further into two modules. Since both modules are free of internal transitions, they could be fused into a single module, a behaviour-equivalent replacement of their parent. The synchronisation labels of the recipient–consumer module are invisible to the root module. We must explicitly define the synchronisation transitions for recv'_m, ign'_m and send'_a to synchronise on the labels recv_m, ign_m and send_a of the root module.

Compared to previous examples, our sliding window protocol model is not very well suited for modular analysis. The internal and external transitions are rarely in conflict, except when the channels have very limited capacity. In fact, synchronisations are possible in almost all local states of the modules.

Table 2. State space sizes (numbers of nodes and edges) for the protocol in Figure 12. The parameters are the transmission and reception window sizes tw and rw and the type of the channel. Each state space is strongly connected. The third columns for modular state spaces indicate the numbers of edges obtained by caching the pre-synchronisation states.

tw,	reliable channel					lossy channel				
rw	flat		modular			flat		modular		
1,1	12	12	8	8	8	108	310	64	172	96
1,2	18	18	12	12	12	462	1,686	348	1,734	636
1,3	24	24	16	16	16	1,336	5,372	1,104	7,928	2,192
2,1	54	90	39	72	54	2,118	8,349	1,422	9,408	2,853
2,2	72	120	52	96	72	9,388	40,256	7,080	65,692	15,156
2,3	90	150	65	120	90	27,265	122,555	22,115	268,185	49,140
3,1	160	336	116	292	192	25,292	113,036	18,412	216,304	40,792
3,2	200	420	145	365	240	109,550	508,790	84,775	1,235,830	193,835
3,3	240	504	174	438	288	323,724	1,537,638	262,026	4,629,366	611,754

Caching Pre-Synchronisation States. Table 2 indicates that applying modular analysis to our model of the sliding window protocol slightly reduces the number of reachable states but increases the number of edges.

The number of edges can be reduced by slightly modifying the procedure $\textsc{Sync}(s_0, \ldots)$ in Figure 5. Before generating the successors of M^*, it would add the item (M^*, tf) to a set and return if the set already contained that item.

Table 2 shows three columns of numbers for modular analysis. The leftmost two columns indicate the numbers of states and edges in the state space of the root net. The third column shows the number of edges reduced by applying this

Table 3. Processor time consumption in seconds and peak heap memory consumption in kilobytes of a 1.67 GHz AMD Athlon XP system running GNU/Linux when exploring the protocol in Figure 12 on lossy channels.

tw,			plain		local cache			pre-cache		pre&local	
rw	nodes	edges	mem	time	mem	time	edges	mem	time	mem	time
1,1	64	172	309	0.0	311	0.0	96	343	0.0	342	0.0
1,2	348	1,734	309	0.1	307	0.0	636	339	0.1	382	0.0
1,3	1,104	7,928	309	0.3	523	0.1	2,192	339	0.3	494	0.1
2,1	1,422	9,408	309	0.3	451	0.1	2,853	339	0.3	422	0.1
2,2	7,080	65,692	309	1.8	2,007	0.5	15,156	343	1.8	2,314	0.5
2,3	22,115	268,185	309	6.0	7,735	2.0	49,140	343	6.0	9,270	1.9
3,1	18,412	216,304	309	4.8	4,475	1.4	40,792	339	4.7	5,754	1.3
3,2	84,775	1,235,830	473	24.2	23,763	8.4	193,835	2,335	24.2	29,914	8.4
3,3	262,026	4,629,366	1,021	82.7	100,679	32.5	611,754	535	84.4	125,290	33.1

method on the root net. The method can only be applied on the root net, since it could prevent synchronisations from occurring in the parent net.

Clearly, this modification does not affect the set of reachable states. However, it is questionable whether the modification is useful in practice. The number of edges obtained by caching the pre-synchronisation states could serve as a benchmark for a more intelligent reduction method.

The times in Table 3 exclude the time needed for invoking a C compiler and linker. The peak memory usage was determined by making Maria allocate everything via `malloc` and by tuning the memory allocator of GNU libc 2.3.1 with `mallopt`. The figures include some non-constant overhead due to pooling. A comparison of the columns "plain" and "pre-cache" reveals only slight differences in resource usage, even though our implementation does not make efficient use of memory, since it stores each pair (M^*, tf) separately.

The effect of this modification depends on implementation aspects and the model. For the interpreter option of Maria, this modification reduces the time needed for exploring the sliding window protocol by 25 %. The model of the automated guided vehicles (Section 5.1) is unaffected by this modification. In the leader election protocol (Table 1), the number of edges is reduced by 20 %.

Caching Local States. Our implementation of the algorithm presented in Figure 5 of Section 4.1 includes an optional synchronisation state cache. Instead of associating a set $\mathcal{S}$ with each invocation of Modules, this option associates a set $\mathcal{S}'$ with each invocation of Explore.

If $(M_{s'}, s', tf, M') \in \mathcal{S}'$, the marking M' of s' is reachable via internal transitions from $M_{s'}$ and a transition synchronising on tf is enabled in M'. The cache maps a local state $(M_{s'}, s')$ to pairs of synchronisation labels and states (tf, M').

The procedure Modules invokes Explore only once for each pair $(M_{s'}, s')$. This caching may save a considerable amount of time if only few of the local successors of $M_{s'}$ are possible synchronisation points. But it will also consume

more memory than the original algorithm, because the cache $\mathbb{S}'$ associated with the root net s_0 must be preserved until the whole model has been explored.

The columns "local cache" and "pre&local" of Table 3 indicate up to 70 % shorter execution times and 200-fold increase in memory usage compared to the columns "plain" and "pre-cache." A more efficient data structure for managing the cache would be needed to apply this modification in practice.

6 Conclusion and Future Work

We presented a slightly more general version of modular high-level nets than Christensen and Petrucci [4] and an algorithm for checking safety properties in these nets by exhaustive enumeration of modular state spaces. Unlike the algorithm sketch of Christensen and Petrucci [5, Section 8.1], our algorithm does not compute any strongly connected components of state space graphs, and thus does not need to store the transition relation. Our implementation consumes only slightly more memory per state than the algorithm for exploring flat state spaces, and it is compatible with the parallel state space exploration option [14].

From the theoretical point of view, it may be difficult to see the benefits of modular analysis. An experienced modeller would eliminate the internal transitions in Figure 1 by rewriting the arc inscriptions of the synchronising transitions and by adapting the initial marking. The flat state space of the resulting model is identical to the modular state space of the root net.

However, just like the precise modelling of complex systems is easier with high-level nets or other high-level languages than with place/transition systems or state machines, we believe that utilising modular analysis can improve productivity. Some of the tedious work of preparing models for verification [18, Section 7.1] can be avoided and shifted to the state space exploration algorithm. It is unnecessary to reduce the number of internal states or to avoid interleavings between local actions in different processes, as the algorithm takes care of them. Unoptimised models are likely to be easier to maintain than optimised ones. Finally, modules or entire models can be reused in other models.

Although we have described the algorithm in terms of Petri nets, we believe that it can be applied to exploring any system that has a notion of processes or modules that communicate via shared actions. It would be interesting to try the algorithm on some real-world high-level models—such as communication protocol specifications written in SDL [9]—and to see whether the results could be improved by applying partial order reduction methods, as in [11].

Section 4.2 lacks an example of a property covering multiple modules, but we believe that such properties can be checked with "assertion" or "fact" synchronisation transitions that should never be enabled. Extending our algorithm to model check liveness properties is the subject of future research.

The specification of modular systems deserves further research. A system can be modularised in several ways. Ideally, the decomposition of the model should

produce greatly reduced state spaces and cater for reuse. Further experiments are needed in order to come up with recommended modularisation strategies.

Acknowledgements. The author would like to thank Charles Lakos for suggesting this research topic, and Laure Petrucci and Charles for valuable feedback during this work. The helpful comments of the anonymous referees are acknowledged.

References

1. Eugenio Battiston, Fiorella De Cindio, and Giancarlo Mauri. Modular algebraic nets to specify concurrent systems. *IEEE Transactions on Software Engineering*, 22(10):689–705, October 1996.
2. Jonathan Billington et al. High-level Petri nets—concepts, definitions and graphical notation, version 4.7.3. Final Draft International Standard ISO/IEC 15909, ISO/IEC JTC1/SC7, Genève, Switzerland, May 2002.
3. Shing Chi Cheung and Jeff Kramer. Checking safety properties using compositional reachability analysis. *ACM TOSEM*, 8(1):49–78, January 1999.
4. Søren Christensen and Laure Petrucci. Modular state space analysis of coloured Petri nets. In Giorgio De Michelis and Michel Diaz, editors, *Application and Theory of Petri Nets 1995, 16th International Conference*, volume 935 of *Lecture Notes in Computer Science*, pages 201–217, Turin, Italy, June 1995. Springer-Verlag.
5. Søren Christensen and Laure Petrucci. Modular analysis of Petri nets. *The Computer Journal*, 43(3):224–242, 2000.
6. Danny Dolev, Maria Klawe, and Michael Rodeh. An $O(n \log n)$ unidirectional distributed algorithm for extrema finding in a circle. *Journal of Algorithms*, 3(3):245–260, September 1982.
7. Orna Grumberg and David E. Long. Model checking and modular verification. *ACM TOPLAS*, 16(3):843–871, May 1994.
8. Gerard J. Holzmann. Spin—formal verification. `http://spinroots.com/`.
9. Specification and description language (SDL). Recommendation Z.100 (08/02), International Telecommunication Union, Geneva, Switzerland, September 2002.
10. Eric Y. T. Juan, Jeffrey J. P. Tsai, and Tadao Murata. Compositional verification of concurrent systems using Petri-net-based condensation rules. *ACM TOPLAS*, 20(5):917–979, September 1998.
11. Bengi Karaçalı and Kuo-Chung Tai. Model checking based on simultaneous reachability analysis. In Klaus Havelund, John Penix, and Willem Visser, editors, *SPIN Model Checking and Software Verification, 7th International SPIN Workshop*, volume 1885 of *Lecture Notes in Computer Science*, pages 34–53, Stanford, CA, USA, August 2000. Springer-Verlag.
12. Orna Kupferman and Moshe Y. Vardi. Model checking of safety properties. In Nicolas Halbwachs and Doron Peled, editors, *Computer Aided Verification 1999, 11th International Conference (CAV99)*, volume 1633 of *Lecture Notes in Computer Science*, pages 172–183, Trento, Italy, July 1999. Springer-Verlag.
13. Marko Mäkelä. Optimising enabling tests and unfoldings of algebraic system nets. In José-Manuel Colom and Maciej Koutny, editors, *Application and Theory of Petri Nets 2001, 22nd International Conference*, volume 2075 of *Lecture Notes in Computer Science*, pages 283–302, Newcastle upon Tyne, England, June 2001. Springer-Verlag.

14. Marko Mäkelä. Efficiently verifying safety properties with idle office computers. In Charles Lakos, Robert Esser, Lars M. Kristensen, and Jonathan Billington, editors, *Formal Methods in Software Engineering and Defence Systems 2002*, volume 12 of *Conferences in Research and Practice in Information Technology*, pages 11–16, Adelaide, Australia, June 2002. Australian Computer Society Inc.
15. Marko Mäkelä. Maria: Modular reachability analyser for algebraic system nets. In Javier Esparza and Charles Lakos, editors, *Application and Theory of Petri Nets 2002, 23rd International Conference*, volume 2360 of *Lecture Notes in Computer Science*, pages 434–444, Adelaide, Australia, June 2002. Springer-Verlag.
16. Laure Petrucci. Design and validation of a controller. In *Proceedings of the 4th World Multiconference on Systemics, Cybernetics and Informatics*, volume VIII, pages 684–688, Orlando, FL, USA, July 2000. International Institute of Informatics and Systemics.
17. Ulrich Stern and David L. Dill. Parallelizing the Murφ verifier. In Orna Grumberg, editor, *Computer Aided Verification 1997, 9th International Conference (CAV97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 256–267, Haifa, Israel, June 1997. Springer-Verlag.
18. Antti Valmari. The state explosion problem. In Wolfgang Reisig and Grzegorz Rozenberg, editors, *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, pages 429–528. Springer-Verlag, 1998.
19. Wei Jen Yeh and Michal Young. Compositional reachability analysis using process algebra. In *Proceedings of the Symposium on Software Testing, Analysis, and Verification*, pages 49–59, Victoria, British Columbia, October 1991. ACM Press.