

International Workshop on
workshop on Petri Nets
Compositions

International Workshop on
Scalable and Usable
Model Checking for Petri Nets
and other models of concurrency

CompoNet 2011 / SUMo 2011

June 21, 2011

Satellite events of PETRI NETS & ACSD 2011
June 20-20, 2010 - Newcastle - UK

Preface

This volume contains the papers presented at the joint SUMo/CompoNet 2011 event: second international workshop on Scalable and Usable Model Checking for Petri nets and other models of concurrency and the first international workshop on Composition of Petri nets held on June 20-24, 2011 in Newcastle, United Kingdom as a part of the International Conference on Applications and Theory of Petri Nets (PETRI NETS 2011). These joint workshops aim at bringing together, in an informal setting, researchers interested in all aspects of model checking and composition for different models of concurrency, whether this interest be practical or theoretical, primary or derived.

SUMo and CompoNet PC members have reviewed nine submissions by researchers from several countries. Each submission was reviewed by 4 program committee members. Finally six papers have been selected for publication and presentation.

The Model Checking Contest is a SUMo's additional event that runs in parallel with the paper submissions, this year the results of this contest will be presented at the conference but not directly published in this proceeding. We expect a future publication of those results after the workshop.

The workshops organizers would like to thank Alexandre Duret-Lutz, from EPITA/LRDE for his Keynote talk entitled "*Building LTL Model Checkers using Transition-based Generalized Büchi Automata*".

The workshops organizers would also like to thank the authors of submitted papers for their interest in SUMo/CompoNet. We also thank the program committee members and the external reviewers for their outstanding work during the reviewing process.

Last but not least, we thank the authors of the EasyChair conference management system which made the practical organization of the reviewing process considerably easier.

June 2011

Didier Buchs (PC chair SUMo)
Hanna Klaudel and Franck Pommereau (PC chairs CompoNet)

Conference Organization

SUMo Programme Committee

Béatrice Bérard (France)	Charles Lakos (Australia)
Dragan Bosnacki (The Netherlands)	Isabella Mastroeni (Italia)
Ivana Cerná (Czech Republic)	Emmanuel Paviot-Adet (France)
Gianfranco Ciardo (USA)	Wojciech Penczek (Poland)
Jean-Michel Couvreur (France)	Denis Poitrenaud (France)
Martin Fränzle (Germany)	Olivier Roux (France)
Giuliana Franchescinis (Italy)	Alexander Serebrenik (Netherland)
Monika Heiner (Germany)	Jeremy Spronston (Italy)
Tommi Junttila (Finland)	Yann Thierry-Mieg (France)
Kais Klai (France)	Bow-Yaw Wang (Taiwan)
Olga Kouchnarenko (France)	Karsten Wolf (Germany)
Marta Kwiatkowska (UK)	

External Reviewers

Edmundo Lopez (University of Geneva)

CompoNet Programme Committee

Eike Best (Germany)	Charles Lakos (Australia)
Didier Buchs (Switzerland)	Johan Lilius (Finland)
Gianfranco Ciardo (USA)	Daniel Moldt (Germany)
Raymond Devillers (Belgium)	Elisabeth Pelz (France)
Berndt Müller (Farwer) (UK)	Laure Petrucci (France)
Alain Finkel (France)	Wolfgang Reisig (Germany)
David de Frutos-Escrig (Spain)	Natalia Sidorova (The Netherlands)
Ryszard Janicki (Canada)	Alex Yakovlev (UK)
Ekkart Kindler (Denmark)	Søren Christensen (Denmark)
Jetty Kleijn (The Netherlands)	Karsten Wolf (Germany)

Table of Contents

Session 1. CompoNet

Distributed Verification of Modular Systems	1
<i>Mohand Cherif Boukala, Laure Petrucci</i>	
Compositional Analysis of Discrete Time Petri nets	17
<i>Yann Thierry-Mieg, Béatrice Bérard, Fabrice Kordon, Didier Lime, Olivier H. Roux</i>	
On the modularity in Petri Nets of Active Resources	33
<i>Vladimir A. Bashkin</i>	

Session 2. SUMo

Optimising the compilation of Petri net models	49
<i>Lukasz Fronc, Franck Pommereau</i>	
Generalized Büchi Automata versus Testing Automata for Model Checking	65
<i>Ala Eddine Ben Salem, Alexandre Duret-Lutz, Fabrice Kordon</i>	
When Graffiti Brings Order	81
<i>Alban Linard, Didier Buchs</i>	

Distributed Verification of Modular Systems

M.C. Boukala¹ and L. Petrucci²

¹ LSI, Computer Science department, USTHB
BP 32 El-Alia Algiers, ALGERIA
boukala@lsi-usthb.dz

² LIPN, CNRS UMR 7030, Université Paris XIII
99, avenue Jean-Baptiste Clément
F-93430 Villetaneuse, FRANCE
Laure.Petrucci@lipn.univ-paris13.fr

Abstract. The use of distributed or parallel processing gained interest in the recent years to fight the state space explosion problem. Many industrial systems are described with large models, and the state space being even larger, it does not fit completely into the memory of a single computer.

To avoid the high space requirement, several reduction techniques have been proposed: modular verification, partial order reductions, symmetries, using symbolic or compact representations like BDDs.

Another way to alleviate the state space explosion problem is to use modular analysis, which takes advantage of the modular structure of a system specification, particularly for systems where the modules exhibit strong cohesion and weak coupling.

In this paper, we propose to combine distributed processing and modular analysis to perform verification of basic behavioural properties such as reachability, deadlock states, liveness, and home states and their distributed analysis for modular systems. Each module is assigned to a process which explores independently the internal activity of the module, allowing a significant reduction in the size of the state space rather than in an interleaved fashion.

Keywords: Modular systems, distributed verification, modular analysis, Petri nets.

1 Introduction

Systems developed nowadays are both more and more complex and critical. When addressing the design of such systems, it is necessary to ensure reliability, by verifying the system properties. The main approach to verification consists in the generation and analysis of the state space. Some properties such as reachability or deadlocks can be verified on-the-fly, and only require information on states reachable from the initial marking. On the contrary, liveness and home states are elaborate properties which require a full generation of the state space including not only nodes but also arcs. Even for small models, the size of the state space may be too huge to fit in the memory of a single computer.

To cope with the state space explosion problem, several reduction techniques have been proposed: on-the-fly verification checks properties during the state space construction; sweep-line construction [Sch04] considers a progress measure, and discards states that will not be encountered further in the construction; modular verification [LP04]; partial order reductions [Val92,NG97]; symmetries [AHI98,CFJ93]; using symbolic or compact representations like BDDs and Kronecker algebra.

Recently, several studies addressed distributed verification, many of them focussing on distributed LTL model-checking [BBS01,BO03,LS99,BP07]. These works are mainly based on the partitioning of the state space.

The performances of distributed verification depends on several criteria, e.g. load balancing of the partitioned state space, but also, more importantly, on a good partitioning. Therefore, choosing an adequate hash function to assign nodes to processors is important.

In [LP04] the modular analysis approach examines in isolation the local behaviour of each subsystem, and then separately considers the synchronisation between the subsystems. In this fashion, exploring the many possible interleavings of activity of the subsystems is avoided, thus reducing the state space.

In this work, we propose to combine modular analysis and distributed processing to improve consequently the systems verification. We focus on checking usual properties such as reachability, liveness and home states.

The paper is organised as follows. We assume the reader is familiar with basic Petri nets notions. Hence, we recall in Section 2 only the modular concepts, i.e. Modular Petri nets and Modular State Spaces. In section 3, we introduce algorithms based on distributed modular construction of the state space. In section 4 distributed verification of various properties is presented. These algorithms have been implemented in a prototype tool and experimental results are presented in section 5. Finally, section 6 concludes the paper.

2 Modular Petri Nets

In this paper, we consider only modules synchronised through shared transitions as in [LP04].

2.1 Definition of Modular Petri Nets

Definition 1 (Modular Petri net). *A modular Petri net is a pair $MN = (S, TF)$, satisfying:*

1. *S is a finite set of modules such that:*
 - *Each module, $s \in S$, is a Petri net: $s = (P_s, T_s, W_s, M_{0_s})$.*
 - *The sets of nodes corresponding to different modules are pair-wise disjoint:*

$$\forall s_1, s_2 \in S : [s_1 \neq s_2 \Rightarrow (P_{s_1} \cup T_{s_1}) \cap (P_{s_2} \cup T_{s_2}) = \emptyset].$$

- $P = \bigcup_{s \in S} P_s$ and $T = \bigcup_{s \in S} T_s$ are the sets of all places and all transitions of all modules.
- 2. $TF \subseteq 2^T$ is a finite set of non-empty transition fusion sets.

In the following, TF also denotes $\bigcup_{tf \in TF} tf$. We now introduce transition groups.

Definition 2 (transition group). A transition group $tg \subseteq T$ consists of either a single non-fused transition $t \in T \setminus TF$ or all members of a transition fusion set $tf \in TF$.

The set of transition groups is denoted by TG .

A transition can be a member of several transition groups as it can be synchronised with different transitions (a sub-action of several more complex actions). Hence, a transition group corresponds to a synchronised action. Note that all transition groups have at least one element.

Next, we extend the arc weight function W to transition groups, i.e. $\forall p \in P, \forall tg \in TG$:

$$W(p, tg) = \sum_{t \in tg} W(p, t), \quad W(tg, p) = \sum_{t \in tg} W(t, p).$$

Markings of modular Petri nets are defined as markings of Petri nets, over the set P of all places of all modules. The restriction of a marking M to a module s is denoted by M_s . The enabling and occurrence rules of a modular Petri net can now be expressed.

Definition 3 (Transition group enabledness). A transition group tg is enabled in a marking M , denoted by $M[tg]$, iff:

$$\forall p \in P : W(p, tg) \leq M(p)$$

When a transition group tg is enabled in a marking M_1 , it may occur, changing the marking M_1 to another marking M_2 , defined by:

$$\forall p \in P : M_2(p) = (M_1(p) - W(p, tg)) + W(tg, p).$$

Figure 1 depicts a modular Petri net consisting of three modules A , B and C . Modules A and B both contain transitions labelled F_1 and F_3 , while modules B and C both contain transition F_2 . These matched transitions are assumed to form three transition fusion sets.

Example: The (full) state space for the modular Petri net of figure 1 is shown in figure 2. Note that the initial state is shown as $A_1B_1C_1$, thus indicating that place A_1 is marked with a token in module A , place B_1 is marked with a token in module B , and place C_1 is marked with a token in module C . In this initial state, only transition F_1 is enabled, its occurrence leading to state $A_2B_2C_1$.

When considering the modular state space, as well as checking properties of the system, we will use Strongly Connected Components. The set of all strongly connected components is denoted by SCC . For a node v and a component $c \in SCC$ we use $v \in c$ to denote that v is one of the nodes in c . A similar notation is used for arcs. We use v^c to denote the component to which v belongs.

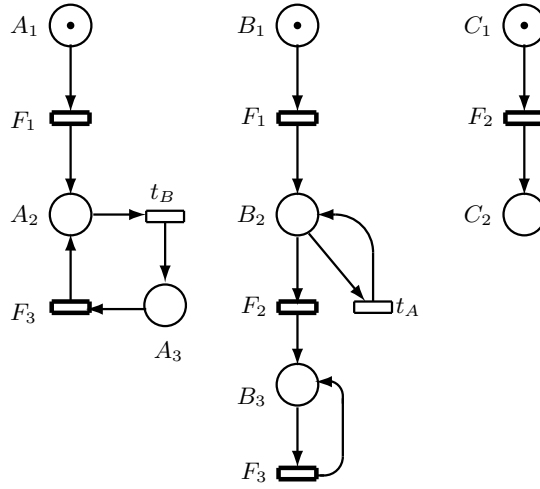


Fig. 1: A modular Petri net with 3 modules

2.2 Modular State Spaces

In the definition of modular state spaces, we denote the set of states reachable from M by occurrences of local (non-fused) transitions only, in all the individual modules, by $[[M]]$.

The notation with a subscript s means the restriction to module s , e.g. $[M]_s$ is the set of all nodes reachable from the global marking M by occurrences of transitions in module s only.

We use $M[[\sigma]]M'$ to denote that M' is reachable from M by a sequence $\sigma \in (T \setminus TF)^*TF$ of internal transitions followed by a fused transition. In the definition of modular state spaces we need a compact notation to capture the states reachable from M in all the individual modules. It turns out that we can use a product of SCCs of the individual modules to express this representative node: for any reachable marking M , we use M^c to denote the product (or tuple) of Strongly Connected Components (SCCs) M_s^c of the individual modules:

$$\forall M \in [M_0] : M^c = \prod_{s \in S} M_s^c.$$

The definition of a modular state space consists of two parts: the state spaces of the individual modules and the synchronisation graph.

Definition 4 (Modular state space). Let $MN = (S, TF)$ be a modular Petri net with the initial marking M_0 . The modular state space of MN is a pair $MSS = ((SS_s)_{s \in S}, SG)$, where:

1. $SS_s = (V_s, A_s)$ is the local state space of module s :
 - (a) $V_s = \bigcup_{v \in V_{SG}} [v]_s$

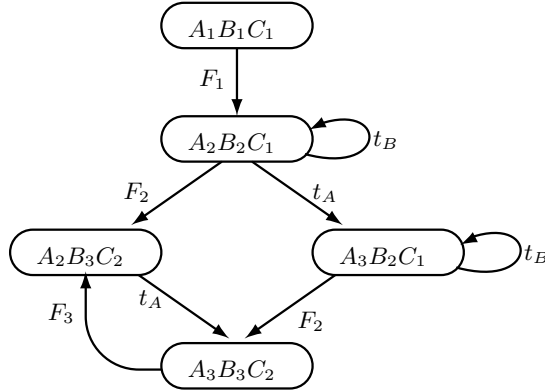


Fig. 2: The state space of the modular net of figure 1

- (b) $A_s = \{(M_1, t, M_2) \in V_s \times (T \setminus TF)_s \times V_s \mid M_1[t]M_2\}$
2. $SG = (V_{SG}, A_{SG})$ is the synchronisation graph of MN:
- (a) $V_{SG} = \llbracket [M_0] \rrbracket^c \cup M_0^c$
- (b) $A_{SG} = \{(M_1^c, (M_1^c, tf), M_2^c) \in V_{SG} \times (\llbracket [M_0] \rrbracket^c \times TF) \times V_{SG} \mid M_1^c \in \llbracket [M_1] \rrbracket \wedge M_1[tf]M_2\}$

A detailed explanation of the definition is given in [LP04]:

(1) The definition of the state space graphs of the modules is a generalization of the usual definition of state spaces.

(1a) The set of nodes of the state space graph of a module contains all states locally reachable from any node of the synchronisation graph.

(1b) Likewise the arcs of the state space graph of a module correspond to all enabled internal transitions of the module.

(2) Each node of the synchronisation graph is labelled by a M^c and is a representative for all the nodes reachable from M by occurrences of local transitions only, i.e. $\llbracket [M] \rrbracket$. The synchronisation graph contains the information on the nodes reachable by occurrences of fused transitions.

(2a) The nodes of the synchronisation graph represent all markings reachable from another marking by a sequence of internal transitions followed by a fused transition. The initial node is also represented.

(2b) The arcs of the synchronisation graph represent all occurrences of fused transitions.

The state space graphs of the modules contain only local information, i.e. the markings of the module and the arcs corresponding to local transitions but not the arcs corresponding to fused transitions. All the information concerning these is stored in the synchronisation graph.

The nodes of the synchronisation graph represent all markings reachable from another marking by a sequence of internal transitions followed by a fused transition. The initial node is also represented.

The arcs of the synchronisation graph represent all occurrences of fused transitions. Each arc is labelled by the corresponding fired transition and by the SCCs of the markings which enabled this transition. But only the SCCs of the participating modules do appear.

Example: The modular state space for the modular Petri net of figure 1 is shown in figure 3. Note that there is a local state space for each module, as well as a synchronisation graph which captures the occurrence of fused transitions. We do not distinguish between nodes and SCCs since, in this case, all SCCs consist of a single node (which is seldom the case in practice).

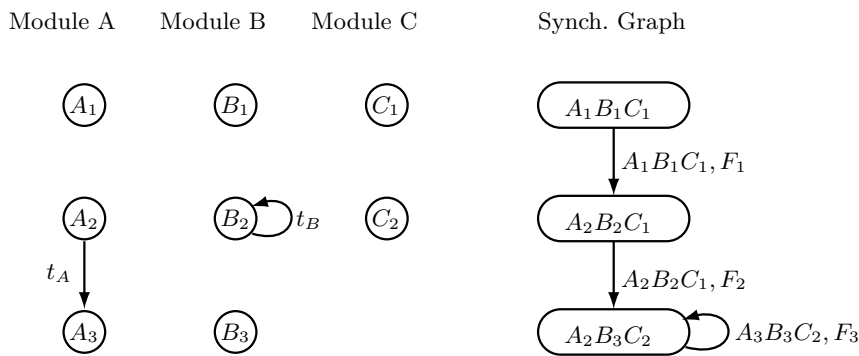


Fig. 3: The modular state space of the net in figure 1

Although sketches of algorithms were introduced in [LP04], we give here the description of the distributed algorithms for the construction of the state space and for the verification of the reachability, dead markings, liveness, and home states properties.

3 Distributed Construction of the Modular State Space

Several works have developed distributed tools generating and exploring the state space on a cluster of workstations. Partitioning the set of states over the different stations is done using a hash function [GMS01, KP04, AAC87, LP04]. This approach can handle larger state spaces but still remains limited.

Here, we consider a different approach based on modularity to achieve a distributed construction of the state space and the verification of various properties.

In this approach we also used two types of processes. They consist in a *coordinator process* and N *worker processes*, one worker process for each module, as in [KP04].

The coordinator constructs the synchronisation graph, coordinates the different worker processes and determines the termination of the modular state space construction, while every worker generates the local state space of the module it is assigned by firing only internal transitions.

The coordinator process starts by sending the initial marking M_{0_s} restricted to module s to each worker process, and waits for receiving the states enabling fused transitions from the workers processes.

Algorithm 1: Internal State Space Generation()

```

1 begin
2   /* Process states in Waiting */
3   while  $\neg$ EndOfGeneration do
4     while  $Waiting \neq \emptyset$  do
5       Choose  $M \in Waiting_s$ 
6       for all the  $t \notin TF$  s.t.  $M[t]M'$  do
7         /*  $t$  is an enabled internal transition */
8         Add ( $M'$ )
9         Arc ( $M, t, M'$ )
10        UpdateSCC ()
11      for all the  $tf \in TF$  s.t.  $M[tf]$  do
12        /*  $M$  enables a fused transition  $tf$  */
13         $TFWaiting \leftarrow TFWaiting \cup \{(M, tf)\}$ ;
14       $Waiting_s \leftarrow Waiting_s \setminus \{M\}$ 
15    if  $TFWaiting = \emptyset$  then
16      EndOfGeneration = true
17    else
18      while  $TFWaiting \neq \emptyset$  do
19        Choose  $(M, tf) \in TFWaiting$  s.t.  $M[tf]M'$ 
20         $Waiting_s \leftarrow Waiting_s \cup \{M'\}$ 
21         $TW.label = M^c$ 
22         $TW.AncSCC = AncSCC(M)$ 
23         $TW.tr = tf$ 
24         $TW.SuccSCC = M'^c$ 
25        Send ( $Sender = s, Dest = Coordinator, Type = SynchTrans,$ 
26               $Content = TW$ )
27      Send ( $Sender = s, Dest = Coordinator, Type = END\_GENERATION$ )
28 end

```

When a worker process s receives the initial marking, it adds it to the sets V_s and $Waiting_s$, that correspond respectively to the set of local markings and the set of markings which are visited but not explored yet. The worker process then invokes algorithm 1 to generate the internal state space of the module it is

assigned. Function `Add` (M) adds the marking M to both sets V_s and $Waiting_s$ if it is not already in V_s .

The workers send enabled fused transitions to the coordinator process. Actually, the message sent to the coordinator not only contains the fused transition tf but also the SCC number M^c of the marking enabling tf , and the SCC number of its predecessor marking in the synchronisation graph (in a field $AncSCC$).

The communications are asynchronous. Thus, when receiving a message, the coordinator and worker processes are preempted and a handler function `MessageHandler()` is invoked. Algorithm 2 describes the actions performed by the coordinator when it receives a fused transition from the worker process s .

The parts handling other messages are not detailed. They concern the algorithm termination (and is similar to the termination in [KP04]), as well as the analysis messages introduced in section 4.

First, the coordinator creates the initial node $v_0 = M_0^c \in V_{SG}$. When a fused transition tf is received from a process s , the successors of a node $v = (v_1, v_2, \dots, v_N) \in V_{SG}$ are computed as follows:

- we consider the set PM of the modules (processes) participating in the synchronisation of the fused transition tf ;
- we construct the sets:
 - W_{pm} corresponding to the fused transition tf received earlier from other processes;
 - W_s corresponding to the fused transition received from process s ;
 - W_{np} contains • for modules not participating in the synchronisation;
- for all possible combinations, the successor nodes in the synchronisation graph are calculated.

The successor states thus computed are sent to the worker processes for an additional round of internal state space generation.

4 Properties

A major issue is the analysis of concurrent systems properties. The reachability graph is the basic model on which most verifications are built. Behavioural properties, which depend on initial marking, and the reachability graph are often used to perform such analysis.

In this work we focus on basic behavioural properties: reachability, deadlocks, liveness, home state and their distributed analysis.

4.1 Reachability

The reachability problem for Petri nets consists in proving that a given marking M is in $[M_0]$. This property is often used to check whether a faulty state M is reachable, e.g. an elevator moving while the door is open. It is convenient to look for erroneous states.

Algorithm 2: Message Handler()

```

1 begin
  :
2   if Message.Type = Synchronisation then
      /* message received contains an enabled fused transition */
3     tf = Message.Content.tr
4     s = Message.Sender
5     PM = {i s.t. tf ∈ Ti}
      /* PM: modules participating in the synchronisation of tf */
6     foreach v = (v1c, v2c, ..., vNc) ∈ VSG do
7       foreach pm ∈ PM and pm ≠ s do
8         Wpm = {w ∈ SGWaitingpm s.t.
9           (w.AncSCC = vpmc) ∧ (w.tr = tf)}
          /* Wpm: markings enabling tf, with vpmc as ancestor */
10      foreach np ∉ PM and np ≠ s do
11        Wnp = {•}
          /* any marking of a non-participating module enables
          synchronisation */
12      Ws = {Message.Content}
13      foreach C = (c1, c2, ..., cN) s.t. ∀i ∈ PM, ci ∈ Wi do
14        v' = (v'1c, v'2c, ..., v'Nc) where:
15          ∀i ∈ PM, v'ic = ci.SuccSCC and ∀i ∉ PM, v'ic = vic
16        Add (v')
17        AddArc (v, (C, tf), v')
18        foreach pm ∈ PM do
19          Send (Sender = Coordinator, Dest = pm, Type = NEW_NODE,
20            Content = v'pmc)
21      SGWaitings = SGWaitings ∪ Message.Content
  :
22 end

```

In some applications, one may be interested in the markings of a subset of places and not care about the others places in the net. This leads to a submarking reachability problem.

Reachability is known to be decidable. For modular systems, determining whether a given state M is reachable from the initial marking can be done in a parallel way: each *worker* process s searches through its local state space V_s . Then, if one of the worker processes does not find M_s in its local state space V_s , marking M is not reachable. Otherwise i.e. $(\forall s \in \{1..N\} M_s \in V_s)$, all the workers send the ancestor SCCs of M_s to the coordinator. The *coordinator* stores the SCCs received from each worker s and checks whether there exists a combination of these in the set of nodes of the synchronisation graph V_{SG} .

4.2 Deadlocks

The algorithm used to find dead markings in a modular state space is based on the following proposition:

Proposition 1 (Deadlocks).

$$M \in [M_0] \text{ is a deadlock} \Leftrightarrow [\forall s \in S : (M_s)^c \in Term(SCC_s) \cap Trivial(SCC_s) \\ \wedge (\forall (v_1, (M_1^c, tf), v_2) \in A_{SG} : M_1^c \neq M^c)]$$

Each worker process s searches in its local state space for the local dead markings which consist in trivial terminal SCCs. If there exists a worker process without such a node, the module assigned to this worker always allows a local behaviour. Hence, the system is deadlock-free. Otherwise, every worker process sends the SCCs corresponding to the locally dead markings to the coordinator process.

The coordinator process stores the SCCs received from the workers. It then checks each combination of such markings: if it labels an arc in the synchronisation graph, the corresponding fused transition is enabled and the marking is not a deadlock. Otherwise, if the marking is effectively reachable, then it is a deadlock.

4.3 Liveness

To verify whether a transition t is live or not, two cases are distinguished: if t is a fused transition ($t \in TF$), or t is an internal transition. The verification is based on the following proposition.

Proposition 2 (Liveness).

1. A transition $tf \in TF$ is live \Leftrightarrow

$$[\forall scc \in Term(SCC_{SG}) : tf \in Trans(scc)] \\ \wedge [\forall v \in V_{SG} : \forall M \in [[v] : (\forall s \in S : M_s^c \in Term(SCC_s)) \\ \Rightarrow \exists (v, (M_1, tf'), v_2) \in A_{SG} : M_1 \in [[M]].]$$

2. A transition $t \in T_s$ is live \Leftrightarrow
- $$\begin{aligned} & [\forall scc \in Term(SCC_{SG}) : \exists v \in scc : t \in Trans([v_s)_s)] \\ & \wedge [\forall v \in V_{SG} : \forall M \in [[v] : (M_s^c \in Term(SCC_s)) \\ & \Rightarrow (t \in Trans(M_s^c) \vee \exists (v, (M_1, tf), v_2) \in A_{SG} : M_1 \in [[M]])]. \end{aligned}$$

To check if a fused transition tf is live, the coordinator checks if there exists a terminal SCC in the synchronisation graph which does not contain transition tf , in which case transition tf is not live. Otherwise, for each node $v \in V_{SG}$ of the synchronisation graph, the coordinator sends v_s to the worker process s and waits to receive the terminal SCCs reachable from v_s in module s . Then, it checks whether the combinations of the terminal SCCs label fused transitions are in SG , considering only the modules participating in the synchronisation of this fused transition. If this is not the case, tf is not live.

When transition t is an internal transition of module s , the worker process s detects the terminal SCCs which do not enable t , that may invalidate liveness. For each node $v \in V_{SG}$ of the synchronisation graph, the coordinator sends v_s to the worker process s . The worker processes check whether these problematic SCCs are reachable from v_s , and send them to the coordinator. The coordinator checks if there exists a combination that does not label a fused transition, in which case transition t is not live.

4.4 Home States

The verification of whether a reachable state M_H is a home state or not is based on the following proposition.

Proposition 3 (Home state).

$$\begin{aligned} & A \text{ state } M_H \in [M_0] \text{ is a home state } \Leftrightarrow \\ & [\forall scc \in Term(SCC_{SG}) : \exists v \in scc : M_H \in [[v]] \\ & \wedge [\forall v \in V_{SG} : \forall M \in [[v] : (\forall s \in S : M_s^c \in Term(SCC_s)) \\ & \Rightarrow M_H \in [[M] \vee \exists (v, (M_1, tf, M_2), v_2) \in A_{SG} : M_1 \in [[M]]]. \end{aligned}$$

To check such a property, we first check whether the state M_H is reachable from all nodes of the synchronisation graph. Then, for every node v of the synchronisation graph, the coordinator asks the worker processes for the terminal SCCs reachable from v by firing internal transitions only. The coordinator checks then that all combinations label an arc, or correspond to the SCC containing M_H . Otherwise M_H is not a home state.

5 Implementation and experiments

The previous algorithms were implemented within a prototype tool. The tests were carried on a cluster composed of 12 stations (Pentium IV with 512 Mbytes of memory), one of them is assigned to the coordinator process while the others run the worker processes.

In this section, we give the results obtained for the *dining philosophers problem* and *Automated Guided Vehicles (AGV) problem*, and draw conclusions.

5.1 The dining philosophers problem

The distributed generation based on partitioning the state space over a cluster of stations, allows for handling large size problems. In [BP07], the philosophers problem was handled with various sizes (see Table 1).

Nb Philo	Nb States	Nb Trans	Nb Cross. arcs	Nb Mess.	CPU Time (sec)
5	11	30	18	36	<0.01
10	123	680	242	494	<0.01
15	1,364	11,310	2,731	5477	0.06
20	15,127	167,240	30,236	60,493	2.52
25	167,761	2,318,400	315,718	631,587	32.58
30	1,728,813	28,686,031	3,572,821	7,156,116	7547.45

Table 1: Distributed generation based on partitioning the state space

But this approach is still limited. The total number of exchanged messages is very high, due to the amount of cross arcs.

In the distributed modular based approach, the original Petri net is split into a Modular Petri net with N modules and each module, which can contain m philosophers, is assigned to a *worker process*. Philosopher i of module l shares its forks with the philosophers $i - 1$ and $i + 1$ of the same module for $2 \leq i \leq m - 1$. Philosophers 1 in module l and m in module $(l - 1) \bmod N + 1$ also share a fork.

The nature of the Petri net gives 4 SCCs in each module graph. The synchronisation graph size is 2^N nodes and $N \cdot 2^N$ arcs. For a problem with 100 philosophers, if we consider 10 modules with 10 philosophers each, we obtain 1,024 nodes and 10,240 arcs in the synchronisation graph, 233 nodes and 1,132 arcs in each local graph. The reachability graph size for the original problem is greater than 7×10^{20} nodes and 7×10^{22} arcs.

In Table 2, we give the modular state space sizes for the philosophers problem considering 10 philosophers per module each time. The average of CPU times of the coordinator is also given.

Nb Philo	Nodes N	Arcs	Mess Nb	CPU Time (sec)	CPU time (1 proc)
20	470	2162	26	0.04	0.08
40	948	4372	52	0.04	0.16
60	1462	6846	84	0.04	0.25
80	2120	10664	120	0.05	0.39
100	3354	21010	230	0.16	0.61

Table 2: Modular distributed generation

The CPU time of the workers is generally equal to 0.04 s (for 10 philosophers per module). Thus, in the first cases the global CPU time corresponds to the

local state space generation, the synchronisation graph generation CPU time is less than 0.01 s. In the last cases the synchronisation graph generation spends more time since its size became larger.

Various tests were performed for reachability, liveness and home state properties, with the philosophers problem of 10 modules with 10 philosophers in each one, the experimental results are given in the table 3.

Properties	CPU Time (sec)	Mess Nb
Reachability1	< 0.01	20
Reachability2	0.03	20
Reachability3	0.02	20
Liveness (<i>Internal tr</i>)	0.17	4278
Liveness (<i>Fuzed tr</i>)	0.15	4116
Home state	0.21	4432

Table 3: Distributed modular verification of properties of the philosophers example

For reachability, we considered three cases: in the first, the marking is not reachable in some modules; in the second, the marking is reachable in the modules but not reachable as a global marking; and in the third case, the marking is reachable. The number of exchanged messages is the same and correspond to messages sent by the *coordinator* to the *worker* processes to transmit the marking to check (10 messages) and to the answers of the workers (10 messages). The liveness and home state verifications are performed with a relatively large number of exchanged messages: since all transitions are live, the coordinator must obtain the terminal SCCs reachable from each node of the synchronisation graph. When the coordinator moves from a node to its successor in the synchronisation graph, it transmits messages only to the workers corresponding to the changed components to have their terminal SCCs. This allows for minimising the number of messages transmitted by the coordinator.

5.2 Automated Guided Vehicles

The Automated Guided Vehicles (AGVs) problem has been solved by means of Modular State Spaces in [LP04]. The problem is that of a factory floor which consists of three workstations which operate on parts, two input and one output stations, and five AGVs which move parts from one station to another.

The 5 AGVs example is loosely coupled. Therefore, much interleaving is avoided when building the modular state space and this leads to very good results. This model has 30,965,760 states (see [LP04]). However, with modular analysis we obtain only 900 states with 2,687 arcs. The analysis based on modular distribution gives also good results as shown in table 4.

	CPU Time (sec)	Mess Nb
State Space Generation	0.02	82
Deadlocks search	0.03	11
Reachability1	< 0.01	22
Reachability2	0.02	22
Reachability3	0.02	22
Liveness (<i>Internal tr</i>)	0.09	3313
Liveness (<i>Fuzed tr</i>)	0.08	3062
Home state	0.15	3415

Table 4: Distributed verification of properties for the 5 AGVs example

6 Conclusion

In this paper, we proposed steps towards distributed modular analysis of Petri net models, based on the construction framework of [LP04]. Using these algorithms, it is possible to verify standard Petri nets properties, such as reachability, deadlocks, home states and liveness, in a distributed manner.

The main advantage of such an approach is the possibility to consider very large systems with several modules, sharing transitions, and assign them among a set of machines, thus limiting the drawbacks of the state space explosion problem.

Each machine (process) generates only the state space of the module it is assigned. The synchronisation graph provides the global knowledge of the behaviour of the system.

Moreover, it is also possible to check properties using the distributed modular state space directly, i.e. without unfolding to the ordinary state space. When designing algorithms there is often a trade-off between time and space complexity. For state space analysis it is attractive to have a rather fast way to decide properties, but the state space explosion problem makes it absolutely necessary to minimise memory usage.

Experiments were performed on a cluster of 12 stations. Both the AGVs, and the philosophers problems were considered, with different sizes. The results obtained for the generation of the modular state space were very interesting and allow for checking properties of very large systems. Future work will extend properties verification to temporal logic properties. Further experiments on larger case studies are also necessary for a better assessment of the benefits of this approach.

References

- [AAC87] S. Aggarwal, R. Alonso, and C. Courcoubetis. Distributed reachability analysis for protocol verification environments. In P. Varaiya and H. Kurzhan-ski, editors, *Discrete Event Systems: Models and Application*, volume 103 of *LNCIS*, pages 40–56. Springer-Verlag, 1987.

- [AHI98] K. Ajami, S. Haddad, and J.-M. Ilié. Exploiting symmetry in linear temporal model checking: One step beyond. In *Proc. 4th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, volume 1384 of *Lecture Notes in Computer Science*, pages 52–67, 1998.
- [BBS01] J. Barnat, L. Brim, and J. Strižbrná. Distributed LTL model checking in SPIN. In *Proc. 8th International SPIN workshop on Model Checking Software (SPIN'2001)*, volume 2057 of *Lecture Notes in Computer Science*, pages 200–216. Springer-Verlag, 2001.
- [BO03] S. Blom and S. Orzan. Distributed state space minimization. *Electronic Notes in Theoretical Computer Science*, 80:1–15, 2003.
- [BP07] M.C. Boukala and L. Petrucci. Towards distributed verification of Petri nets properties. In British Computer Society eWIC, editor, *In Proc. 1st International Workshop on Verification and Evaluation of Computer and Communication Systems (VECOS'07), Algiers, Algeria*, pages 15–26, 2007.
- [CFJ93] E. M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. In *Proc. 5th Int. Conf. Computer Aided Verification (CAV'93)*, volume 697 of *Lecture Notes in Computer Science*, pages 450–462. Springer-Verlag, 1993.
- [GMS01] H. Garavel, R. Mateescu, and I. Smarandache. Parallel state space construction for model-checking. In *Proc. 8th International SPIN workshop on Model Checking Software (SPIN'2001)*, volume 2057 of *Lecture Notes in Computer Science*, pages 217–234. Springer-Verlag, 2001.
- [KP04] L. Kristensen and L. Petrucci. An approach to distributed state exploration for coloured Petri nets. In *Proc. 25th Int. Application and Theory of Petri Nets (ICATPN'2004)*, volume 3099 of *Lecture Notes in Computer Science*, pages 474–483. Springer-Verlag, 2004.
- [LP04] C. Lakos and L. Petrucci. Modular analysis of systems composed of semiautonomous subsystems. In *In Proc. 4th Int. Conf. on Application of Concurrency to System Design (ACSD'2004), Hamilton, Canada.*, pages 185–194. IEEE Computer Society Press, 2004.
- [LS99] F. Lerda and R. Sisto. Distributed-memory model checking with SPIN. In *Proc. 5th and 6th International SPIN workshops on Model Checking Software (SPIN'1999)*, volume 1680 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, 1999.
- [NG97] R. Nalumasu and G. Gopalakrishnan. A new partial order reduction algorithm for concurrent systems. In *Proc. Int. Conf. Hardware Description Languages and their Applications (CHDL'97)*. Chapman & Hall, 1997.
- [Sch04] K. Schmidt. Automated generation of a progress measure for the sweep-line method. In *Proc. 10th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *Lecture Notes in Computer Science*, pages 192–204. Springer-Verlag, 2004.
- [Val92] A. Valmari. A stubborn attack on state explosion. *Formal Methods in Systems Design*, 1(4):297–322, 1992.

Compositional Analysis of Discrete Time Petri nets

Yann Thierry-Mieg¹, Béatrice Bérard¹, Fabrice Kordon¹, Didier Lime², and Olivier H. Roux²

¹ Université Pierre & Marie Curie, LIP6/MoVe
4, place Jussieu, F-75252 Paris CEDEX 05, France

² LUNAM Université, École Centrale de Nantes & IRCCyN - UMR CNRS 6597
1, rue de la Noë, B.P. 92101, 44321 Nantes Cedex 3, France

Abstract. Symbolic BDD-based verification techniques successfully tackle combinatorial explosion in many cases. However, the models to be verified become increasingly larger and more complex, including - for instance - additional features like quantitative requirements and/or a very high number of components. The need to improve performances for verification tools thus remains a challenge. In this work, we extend the framework of Instantiable Transition Systems in order to (i) take into account time constraints in a model and (ii) capture the symmetry of instances which share a common structure, thus significantly increasing the power of our tool. For point (i), we implement timed models with discrete time semantics and for (ii), we introduce scalar sets as a special form of composition. We also report on experiments including comparisons with other tools. The results show a good scale up for our approach.

1 Introduction

Context. Model checking is now widely used as an automatic and exhaustive way to verify complex systems. However, this approach suffers from an intrinsic combinatorial explosion, due to both a high number of synchronized components and a high level of expressivity in these components.

Among the different methods proposed to tackle the problem, using decision diagrams [8] or partial order based techniques [21] have proved successful. Moreover, exploitation of symmetries [13] or compositional model checking [2, 16] can be most successful, especially when several components share the same structure (like the train models in the train crossing example or the processes in Fischer's protocol).

With respect to the expressivity issue, we consider the particular problem of introducing explicit time constraints in the components of a system. In this modeling step, the choice of a time domain is important, impacting on the size of the resulting model, the class of properties which can be verified and the performances of the verification.

During the last twenty years, numerous variants of dense time models have been extensively studied. Among them, Time Petri Nets [19] (TPN) and networks of Timed Automata [1] (TA) benefit from verification tools, which implement techniques relying on the construction of a class graph (for TPN) or zone graph (for TA), with dedicated and efficient data structures to represent zones, like Difference Bounded Matrices [10] (DBM), which have been used in both cases.

If we now consider a growing number of timed components, the two problems of expressivity and size occur together, but the symbolic DBM encoding only applies to time zones and does not concern the discrete part of the states. Furthermore, the union of DBM may not be convex, thus cannot be encoded as a DBM, so mixing DBM technology with a symbolic encoding of discrete states is difficult.

Directly interpreting models over a discrete time domain is usually easier to handle than dense time because mechanisms elaborated for model checking of discrete systems can be reused, even though state space explosion can be worse. The relations between dense and discrete time analysis have been discussed for various models, showing that, in many cases, discrete time computation is sufficient to preserve reachability or time-bounded properties [14, 22]. Contrary to common belief, while the discretized approach is sensitive to the maximum clock values in a model, it can often outperform dense time approaches.

Contribution. In this work, we modify and extend the framework of Instantiable Transition Systems proposed in [24] to include two new features. The first one is a special operation of composition, building what we call *scalar sets* to capture symmetries in components sharing the same structure. The second one is the implementation of discrete time semantics for Time Petri Nets, following the latter approach to propose a new fully symbolic technique for reachability analysis of discrete time specifications. The new tool Roméo/SDD thus relies on hierarchical set decision diagrams (SDD [9]) that offer state of the art automatic symbolic saturation algorithms [12].

We experiment on two classical benchmark examples combining both features, and compare the performances with those of several other tools handling discrete or dense time, showing gains of several orders of magnitude.

Outline. We first recall the definition and semantics of Time Petri Nets in Section 2, along with the train crossing example. Section 3 describes the Instantiable Transition Systems framework with the additional features and briefly presents the encoding technique. Finally, in Section 4, we give the performances obtained with our prototype implementation, with comments and comparisons.

2 Discrete time Petri nets

To handle time constraints, we propose to use discrete time models. As mentioned in the introduction, this can lead to larger state spaces due to sensitivity to constants. However, the main advantage of this approach is to reduce the problem of timed verification to a plain event-based verification.

Let us first recall the classical definition of Discrete Timed Transition Systems (DTTS), where all standard actions (from some set A) are considered instantaneous and delay transitions are added, in a time domain restricted to the set \mathbb{N} of natural numbers. Without loss of generality, we consider only delay steps of exactly one time unit (special action $\mathbb{1}$ below). Having a single basic operation to handle time delay is more effective in a symbolic setting than attempting to find at each step the maximal integer delay consistent with synchronization of several components in a set of states.

Definition 1 (DTTS). *Let A be a set of action labels and let $\mathbb{1} \notin A$ be a special action representing a one time unit delay. A Discrete Timed Transition System over A is a*

tuple $\mathcal{T} = \langle S, s_0, A, \rightarrow \rangle$ where S is a set of states, $s_0 \in S$ is the initial state, and $\rightarrow \subseteq S \times (A \uplus \{\mathbb{1}\}) \times S$ is the transition relation (\uplus stands for disjoint union).

We consider the model of Time Petri Nets (TPN) with discrete time semantics. TPN are used to compactly model concurrent timed behaviors. Besides, regarding the problem of marking reachability, discrete time semantics capture all possible behaviors [22, 18], even those with dense time semantics, which makes it possible to compare experimentation results in both cases.

We choose an extended definition of TPN because this leads to easier and more compact modeling abilities. There is a quite strong community of extended TPN users and our definition below captures a wide superset of what is understood as TPN in the literature: we consider an enabling predicate and a firing function as syntactic requirements, instead of defining various sorts of arcs. This homogeneously subsumes extensions such as reset arcs, read (or test) arcs, inhibitor arcs, and even non-deterministic extensions like hyper-arcs (because *fire* maps to $2^{\mathbb{N}^{Pl}}$), which are offered for instance by the Roméo tool [11]. The rich formalism demonstrates the flexibility of our tool, which supports arbitrary models with (finite) DTTS semantics.

Definition 2 (TPN). A Time Petri Net is a tuple $\mathcal{N} = \langle Pl, Tr, A, enabled, fire, \ell, m_0, \alpha, \beta \rangle$ where:

- Pl is a finite set of places, Tr is a finite set of transitions (with $Pl \cap Tr = \emptyset$),
- A is a finite set (alphabet) of action labels which contains a distinguished local label \top ,
- $enabled : \mathbb{N}^{Pl} \times Tr \mapsto \{true, false\}$ is an enabling predicate, $fire : \mathbb{N}^{Pl} \times Tr \mapsto 2^{\mathbb{N}^{Pl}}$ is a transition firing function, $\ell : Tr \mapsto A$ is a labeling function,
- $m_0 \in \mathbb{N}^{Pl}$ is the initial marking of the net,
- $\alpha : Tr \mapsto \mathbb{N}$ and $\beta : Tr \mapsto \mathbb{N} \cup \{\infty\}$ are functions satisfying $\forall t \in Tr, \alpha(t) \leq \beta(t)$ called respectively earliest (α) and latest (β) transition firing times.

For instance, standard Place/Transition nets are usually defined using pre (noted **Pre**) and post (noted **Post**) functions : $Pl \times Tr \mapsto \mathbb{N}$. Then, for a marking $m \in \mathbb{N}^{Pl}$ and a transition $t \in Tr$, enabling is defined by $enabled(m, t)$ iff $\forall p \in Pl, m(p) \geq \mathbf{Pre}(p, t)$ and transition firing by $fire(m, t) = \{m'\}$ with $\forall p \in Pl, m'(p) = m(p) - \mathbf{Pre}(p, t) + \mathbf{Post}(p, t)$. Inhibitor arcs **Inh** and test arcs **Test** are defined similarly and add enabling conditions to a transition: $enabled(m, t)$ iff $\forall p \in Pl, m(p) < \mathbf{Inh}(p, t) \wedge m(p) \geq \mathbf{Test}(p, t)$. Note that the *enabling* predicate only considers markings while timing conditions are defined separately. In definition 2, transitions are equipped with labels for further composition of nets (see Section 3).

The classical **train crossing example** [5] is partly described by the three TPNs in Fig. 1. Each train triggers a sensor App when approaching the critical zone, then takes 3 to 5 time units to reach the crossing, and 2 to 4 time units to leave it, where it triggers the Exit sensor. The controller of Fig. 2 keeps track of how many trains are in the critical zone (this model considers n tracks go through the gate, so up to n trains could be in the zone). It strives to identify when the first train enters the area, or when the last train leaves it, essentially by counting trains as they trigger App and Exit sensors. Finally the gate itself is modeled as a TPN which reacts to control commands (App and Exit) but

with some delays introduced to model the time it takes to open or close the gate. These three TPN models are building blocks we will assemble to build the full system model; this allows us to easily build variations on a model while reusing parts of a model in different scenarios.

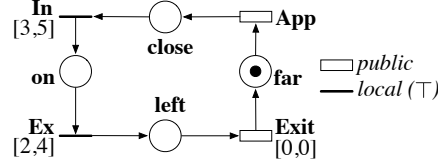


Fig. 1. Train Component (default time interval is $[0, \infty[$)

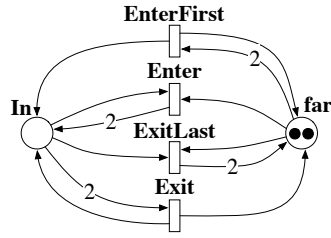


Fig. 2. Controller module for 2 trains

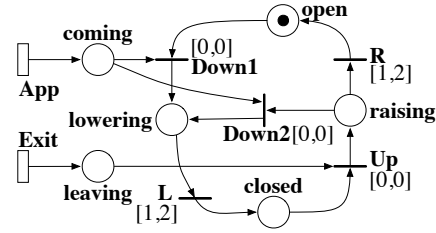


Fig. 3. Gate module (default time interval is $[0, \infty[$)

The discrete time semantics of a TPN is described by a Discrete Time Transition System (DTTS). A *valuation* v is an element of \mathbb{N}^T . Thus, for a transition $t \in T$, $v(t)$ represents the value in \mathbb{N} of an implicit clock associated with t .

Definition 3 (Discrete Time Semantics of a TPN). For a Time Petri Net $\mathcal{N} = \langle Pl, Tr, A, enabled, fire, \ell, m_0, \alpha, \beta \rangle$, the semantics is a transition system $S_{\mathcal{N}} = \langle S, s_0, A \cup \{\mathbb{1}\}, \rightarrow \rangle$ where:

- $S = \mathbb{N}^{Pl} \times \mathbb{N}^{Tr}$ is the set of states. An element s of S is a pair $s = \langle m, v \rangle$, where m is the place marking and v is the transition clock valuation.
- $s_0 = \langle m_0, \bar{0} \rangle$, where the tuple $\bar{0}$ corresponds to the value 0 for all transition clocks.
- $\rightarrow \subseteq S \times (A \uplus \{\mathbb{1}\}) \times S$ is the transition relation defined for states $\langle m, v \rangle, \langle m', v' \rangle$ by:

The **discrete** transition relation:

$\langle m, v \rangle \xrightarrow{a} \langle m', v' \rangle$ iff there is a transition $t \in Tr$ such that

$$\left\{ \begin{array}{l} \ell(t) = a \wedge enabled(m, t) \wedge v(t) \geq \alpha(t) \\ \text{and } m' \in fire(m, t) \\ \text{and } \forall t' \in Tr, v'(t') = \begin{cases} v(t') & \text{if } enabled(m', t') \wedge t \neq t' \\ 0 & \text{otherwise} \end{cases} \end{array} \right.$$

The **delay** transition relation:

$$\langle m, v \rangle \xrightarrow{\mathbb{1}} \langle m', v' \rangle \text{ iff } m' = m \text{ and for all } t \in Tr,$$

$$v'(t) = \begin{cases} \text{enabled}(m, t) \implies v(t) < \beta(t) & (\text{urgent clocks prevent elapse}) \\ \begin{cases} v(t) + 1 \text{ if } \text{enabled}(m', t) \wedge \beta(t) \neq \infty & (\text{normal elapse}) \\ v(t) + 1 \text{ if } \text{enabled}(m', t) \wedge \beta(t) = \infty \wedge v(t) < \alpha(t) & (\text{only progress up to } \alpha) \end{cases} \\ v(t) \text{ otherwise} \end{cases}$$

Note that we use here *atomic* semantics and a *newly disabled* criterion to reset disabled transition clocks, ensuring a disabled transition has a clock value set to 0.

The rule (*only progress up to α*) allows us to handle the infinity problem due to unbounded latest firing times (e.g. for transitions with firing interval $[\alpha(t), \infty]$). With this strategy, we do not let clocks of the corresponding transitions progress up to more than their lowest significant value $\alpha(t)$. The behavior can thus be accurately represented on a finite support. If the logic used to express properties involves atomic propositions with test of clock values, the rule can be relaxed to let the clock progress be tracked up to the highest value tested in the logic, rather than $\alpha(t)$.

3 Instantiable Transition Systems

This section defines Instantiable Transition Systems (ITS), a framework designed to exploit the hierarchical characteristics of SDD [9], the data structure used in the tool to encode the state space, for the description of component based systems. ITS were introduced in [24], but the definitions below are more expressive. In particular, *multisets* over an alphabet of action labels are replaced by *words*. We then introduce additional ITS types, to build "regular" composite types and to capture the semantics of (discrete) timed models.

3.1 ITS types, instances and composites

ITS describe a minimal Labeled Transition System (LTS) style formalism using notions of *type* and *instance* to emphasize locality of actions and to exploit the similarity of instances of a given type. The composition mechanism is based solely on transition *synchronizations* (no explicit shared memory or channel).

Notations: The set of finite words over a finite alphabet A is denoted by A^* , with ϵ for the empty word and \cdot (or no symbol) for the concatenation operation. We denote by $z.X, z.Y \dots$ the element X (resp. $Y \dots$) of a tuple $z = \langle X, Y, \dots \rangle$.

Definition 4 sets an abstract contract or interface that must be implemented by concrete ITS types.

Definition 4 (ITS Semantics). An ITS type is a tuple $\tau = \langle S, A, Locals, Succ \rangle$ where:

- S is a set of states; A is a finite set of public action labels;
- $Locals : S \mapsto 2^S$ is a local successors function;
- $Succ : S \times A^* \mapsto 2^S$ is a transition function satisfying: $\forall s \in S, Succ(s, \epsilon) = \{s\}$.

Let T be a set of ITS types. An **ITS instance** i is defined by an ITS type $\text{type}(i) \in T$ and a set of states $\text{type}(i).S$.

Reachability: A state s' is reachable in an instance i from the state s_0 iff $\exists s_1, \dots, s_n \in \text{type}(i).S$ such that $s' = s_n$ and $\forall 1 \leq j \leq n, s_j \in \text{type}(i).Locals(s_{j-1})$.

The two functions *Locals* and *Succ* are used for different purposes: *Locals* represents moves that may occur within an instance autonomously or independently from the rest of the system. Hence it returns states reachable through occurrences of local events. The function *Succ* produces successors by explicitly synchronizing actions via a word over the alphabet of action labels. Synchronizing on an empty word leaves the state of the instance locally unchanged. Note that *Succ* is the only way to control the behavior of a (sub)system from outside.

Remark 1. The transition relation of a full system can only be defined in terms of subsystem synchronizations using *Succ* and of independent local behaviors. A full system is defined by a single instance of a particular type in a specific initial state. Because it is self-contained (there is no notion of environment that could trigger *Succ*) reachability only depends on the definition of *Locals*.

Remark 2. Apart from distinguishing the special time delay action label $\mathbb{1}$, a DTTS is thus simply a labeled transition system and is immediately compatible with the ITS framework, if the DTTS has a local label \top . Interpreting timed models such as Time Petri Nets (but also Timed Automata) over discrete time makes it possible to use the ITS model-checking engine, and also profit from the Composite and Scalar set definitions below to build compositional models.

We now define a *composite ITS type*, designed to offer support for the hierarchical composition of ITS instances. A version of a composite type was presented in [24] but it introduced more syntactic elements, and was less expressive than the version presented here. This version is aligned with standard labeled synchronized product definitions (e.g [3, 16]), with the addition of the possible aggregation of several steps into an atomic transition sequence, by a word composed of action labels.

Notations: Given a cartesian product $Z = Z_1 \times \dots \times Z_n$ of sets Z_1, \dots, Z_n , we denote by π_i the projection operator $Z \mapsto Z_i$. For a set $I = \{i_1, \dots, i_n\}$ of ITS instances (where an arbitrary order is chosen), S_I is the set $\text{type}(i_1).S \times \dots \times \text{type}(i_n).S$ and A_I is the set $\text{type}(i_1).A^* \times \dots \times \text{type}(i_n).A^*$. Cardinality of I is denoted by $|I|$.

Definition 5 (Composite). A *composite* is a tuple $C = \langle I, \text{Sync}, A, \lambda \rangle$ where:

- I is a finite set of ITS instances, said to be contained by C . We further require that the type of each ITS instance already exists when defining I , in order to prevent circular or recursive type definitions.
- $\text{Sync} \subset A_I$ is the finite set of synchronizations; A is a set of action labels, which contains the label \top and $\lambda : \text{Sync} \mapsto A$ is the labeling function

Notations: The next state function $\text{Next}_I : S_I \times A_I \mapsto 2^{S_I}$, used in definition 6 below, is defined for $s, s' \in S_I$ and $\sigma \in A_I$ by:

$$s' \in \text{Next}_I(s, \sigma) \text{ iff } \forall i \in I, \pi_i(s') \in \text{type}(i).Succ(\pi_i(s), \pi_i(\sigma))$$

Definition 6 (ITS Semantics of a Composite). The ITS type $\tau = \langle S, A, Locals, Succ \rangle$ corresponding to a composite $C = \langle I, Sync, A', \lambda \rangle$, is defined by:

- $S = S_I ; A = A' \setminus \{\top\}$;
- $Locals : S \mapsto 2^S$ is defined for $s, s' \in S$ by: $s' \in Locals(s)$ iff

$$\begin{cases} \exists i \in I, \pi_i(s') \in type(i).Locals(\pi_i(s)) \wedge \forall j \in I, j \neq i, \pi_j(s') = \pi_j(s) \\ \text{or } \exists \sigma \in Sync, \lambda(\sigma) = \top, s' \in Next_I(s, \sigma) \end{cases}$$
- $Succ : S \times A^* \mapsto 2^S$ is defined for $s, s' \in S, w = a_1 \cdots a_n \in A^*$ by: $s' \in Succ(s, w)$ iff $\exists \sigma_1, \dots, \sigma_n \in Sync, \exists s_0, \dots, s_n \in S, \forall j \in [1..n], \lambda(\sigma_j) = a_j \wedge s_j \in Next_I(s_{j-1}, \sigma_j) \wedge s_0 = s \wedge s_n = s'$.

Definition 6 thus describes an implementation of the generic ITS type contract. It contains either elementary instances (such as LTS, or as we will use later in this paper, a discrete timed transition system), or inductively other instances of composite nature.

$Locals(s)$ is defined as the set of states resulting from the action of $Locals$ in any nested instance (without affecting the other instances), or states reachable from s through the occurrence of any synchronization associated to the local label \top .

$Succ(s, w)$ is obtained by composing the effects of each label a in the word w using the cartesian product. So the use of a word can force system progression by firing several action labels in an atomic sequence. Firing an action label corresponds to arbitrarily choosing a synchronization that bears this label, and firing it.

$t_0: \text{train}$	$t_1: \text{train}$	λ	$tg: \text{TrainGroup}$	$cc: \text{ContrGate}$	λ	$g: \text{gate}$	$c: \text{controller}$	λ
App	ε	App	Exit	Exit	\top	App	EnterFirst	App
ε	App	App	App	App	\top	Exit	EnterLast	App
Exit	ε	Exit	$\mathbb{1}$	$\mathbb{1}$	$\mathbb{1}$	ε	Exit	Exit
ε	Exit	Exit				$\mathbb{1}$	$\mathbb{1}$	Exit
$\mathbb{1}$	$\mathbb{1}$	$\mathbb{1}$				$\mathbb{1}$	$\mathbb{1}$	$\mathbb{1}$

Fig. 4. Synchronization for 2 trains
Fig. 5. Synchronization for the complete system
Fig. 6. Synchronization for a 2 train controller and a gate

Figures 4, 5 and 6 show the composite types used to model the train crossing example. For instance, let us consider in Figure 4 a representation of a composite ITS type. It contains two instances t_0 and t_1 of an ITS type “Train”, and defines five synchronizations (lines in the table) and three labels **App**, **Exit** and $\mathbb{1}$. A state s of this composite is thus defined as a cartesian product of the state of instance t_0 (noted $\pi_{t_0}(s)$) and t_1 . The successors obtained by $Succ(s, \text{App})$ are the states in which either t_0 or t_1 have fired **App** and the state of the other instance is unchanged (e.g. s' such that $\pi_{t_0}(s') \in Train.Succ(\pi_{t_0}(s))$ and $\pi_{t_1}(s') = \pi_{t_1}(s)$ or vice versa). There is no local (\top labeled) synchronization in this example, thus successors by $Locals$ are states in which either t_0 or t_1 have progressed by $Train.Locals$.

Although this tabular presentation for composite is directly linked to the formal definition, Roméo/SDD uses a graphical syntax to express how instances are connected, which is more user-friendly.

3.2 Regular Models

While the definition of an ITS composite permits hierarchical modeling, the notion of *Scalar Set* ITS type, where the synchronizations are defined in a parametric way, deals with “regular” or symmetrically composed systems. This definition is not more expressive than the one for a composite but it allows us to build several equivalent composite representations of a system (see Figures 7 and 8), with a possible impact on performances. We thus offer a way of describing symmetric models, so that the manually built recursive encodings presented in [24] can be easily applied to symmetric problems. The scalar set captures a frequent symmetric synchronization pattern when using a set of identical instances and its definition is the same as those proposed in symmetric Uppaal [17], Murphi [15] or in symmetric nets.

Definition 7 (Scalar Set). A *scalar set* is a tuple $S = \langle \tau, n, s_0, D, CSync \rangle$ where:

- $\tau = \langle S, A, Locals, Succ \rangle$ is the ITS type of the contained instances.
- $n \in \mathbb{N}$ is the the number of instances
- $s_0 \in \tau.S$ is the initial state of the instances
- D is a subset of $\tau.A \times \{ANY, ALL\} \times \{public, private\}$, called the set of delegates,
- $CSync$ is a subset of $\tau.A \times \tau.A \times \{public, private\}$, called the set of circular synchronizations.

A scalar set can thus be seen as a subclass of composite, containing n identical instances of a type τ , and offering only two ways of synchronizing them, *ANY* and *ALL*. A delegate $d = \langle a, t, v \rangle$ of type $t = ANY$ affects exactly one of the contained instances, chosen arbitrarily. In other words, an *ANY* delegate maps to n synchronization lines: each line affects a single instance with action a . In contrast, a delegate of type $t = ALL$ targets all contained instances simultaneously, and maps to a single synchronization line of as . The *visibility* v of a delegate gives the labeling function of the composite: the label of the resulting synchronization lines is \top if *private* is used, or action a otherwise.

For instance, the train group model for 2 trains (see Figure 4) can be expressed as the following scalar set:

$$\langle Train, 2, s_0, \{ \langle App, ANY, public \rangle, \langle Exit, ANY, public \rangle, \langle \mathbb{1}, ALL, public \rangle \}, \emptyset \rangle.$$

A scalar set represents a regular model pattern and produces a homogeneous representation of parametric models. Furthermore, because this pattern is very constrained, different semantically equivalent encodings can be considered at the SDD level. In particular, as introduced in [24], recursive encodings can be used. For instance, the train group model for 4 trains can be represented as a composite of 4 trains, or a composite containing 2 instances of a composite with two trains (Figures 7 and 8).

Circular synchronizations (*CSync* in Def. 7) capture another frequent composition pattern: topological rings, where a component synchronizes with its successor in the ring. For instance, Dijkstra’s classical dining philosophers example for N philosophers can be written as:

$$\langle PhiloFork, N, s_0, \emptyset, \{ \langle getFork, getLeft, private \rangle, \langle putFork, finishEating, private \rangle \} \rangle.$$

In this set, *PhiloFork* is a composite of a Fork and a Philosopher, *getFork* and *putFork* are actions that take or return the fork, *getLeft* and *finishEating* are actions where the Philosopher acquires or releases his left neighbor’s fork.

t_0 : train	t_1 : train	t_2 : train	t_3 : train	λ
App	ε	ε	ε	App
ε	App	ε	ε	App
ε	ε	App	ε	App
ε	ε	ε	App	App
Exit	ε	ε	ε	Exit
ε	Exit	ε	ε	Exit
ε	ε	Exit	ε	Exit
ε	ε	ε	Exit	Exit
$\mathbb{1}$	$\mathbb{1}$	$\mathbb{1}$	$\mathbb{1}$	$\mathbb{1}$

Fig. 7. Flat representation of 4 train scalar set.

t_0 : train2	t_1 : train2	λ
App	ε	App
ε	App	App
Exit	ε	Exit
ε	Exit	Exit
$\mathbb{1}$	$\mathbb{1}$	$\mathbb{1}$

Fig. 8. Recursive representation of 4 train scalar set, as two times two trains. The type "train2" corresponds to the composite of figure 4

In [24], several strategies were manually experimented to encode such regular models, the most basic one building a composite containing n instances of the embedded type. This can be generalized by building a composite of n/k instances of a composite containing k instances (or $k + 1$ to capture the remainder of the division n/k) of the basic type. More subtle are recursive encoding strategies, where the type of a (sub-)composite containing k instances is itself defined as a group of groups of instances. This recursive strategy leads in some cases (like for the dining philosophers) to logarithmic overall complexity in time and memory.

With these additional definitions of scalar set, the encoding strategy can be configured by the user at run time, by simply setting an option. Two parameters guide the encoding: The width gives the number of variables at any given level of composite, and the depth gives the number of levels of hierarchy or nesting introduced. The user can choose to bound one or the other and select the more efficient. For instance the flat encoding of Fig. 7 has width 4 and depth 1, while the encoding of Fig. 8 has width 2 and depth 2.

We thus generalize for easy reuse the very favorable encodings from [24] (for un-timed systems), which thanks to hierarchy can be exponentially more efficient than what is available with other decision diagram variants.

3.3 ITS Tools

The ITS tools can be used for modeling and analysis of ITS specifications. The graphical front-end is an Eclipse plugin built upon Coloane (configure **coloane.lip6.fr/night-updates** in eclipse update sites), thus runs on all platforms. The actual analysis tools are provided on **ddd.lip6.fr** as pre-compiled binaries for common platforms (Linux, MacOS, Windows).

In the modeling environment, TPN can be used as building bricks to define ITS instances. The tool currently features import/export functionality for both Roméo and Tina formats, full modeling capability, the ability to "flatten" a composite ITS definition to an equivalent TPN, use of variables in arc labels and time constraints, and CTL model-checking for analysis. To jump-start new users, both examples used in this paper are available directly through the "New->Example" eclipse menu. Figure 9 shows


```

if  $t_i$  enabled then
  if  $clock(t_i) < t_i.lft$  then
    | increment  $clock(t_i)$ 
  else
    | return  $\emptyset$ 
  end
else
  | return  $Id$ 
end

```

Algorithm 1: Part of the encoding for $\mathbb{1}$ transition.

a screenshot of the modeling tool, where composite and scalar types use a graphical syntax.

The analysis engine of the tool uses the powerful hierarchical set decision diagram (SDD) technology. SDD are a variant of reduced decision diagrams producing a compact representation of very large state spaces. Their main characteristic exploited in this work is that edges of the decision diagram can bear references to SDD, allowing a hierarchical encoding of the state-space.

The semantics of TPN and composite ITS are directly expressed by operations acting on the SDD. The delay transition $\mathbb{1}$ uses a conjunction (over all transitions) of if-then-else constructs to increment the currently enabled clocks or forbid time elapse if a latest firing time has been reached. An informal presentation is given for this $\mathbb{1}$ operation on transition t_i by Algorithm 1: returning \emptyset indicates there are no successors, while returning Id indicates that with respect to this transition no updates to the current state are needed. This algorithm is slightly adapted to take into account infinite latest firing times, as explained in the discrete TPN semantics paragraph above.

4 Experiments and comparisons to related work

Table 1 reports our results collected on a 1.83GHz Intel Xeon, with 4GB of RAM. Two classical benchmark problems from the literature [4, 25, 17] are modeled using ITS: Berthomieu’s version of the train-gate controller with multiple tracks and Fischer’s mutual exclusion protocol (described below). These models are parameterized and thus easily scalable for benchmarking.

Fischer’s protocol. This protocol is modeled using two elementary TPN. We use reset (respectively read) arcs, according to their classical definition, which means they allow to reset (resp. check the non-emptiness) the marking of the associated place. The process type (Fig.10) represents the behavior of a single process. The resource type (Fig. 11) is used to block processes in the *idle* state during the execution of the protocol (*i.e.* at least one process has already reached *wait* or *cs*).

Then, we build a process group (Fig. 12) containing a scalar set of processes in the system, similarly to the approach used for the train group. Finally, this group of processes is composed with an instance of the resource according to the synchronizations defined in Fig. 13. Note (line 3) the use of a word in A^* for synchronisation: for a

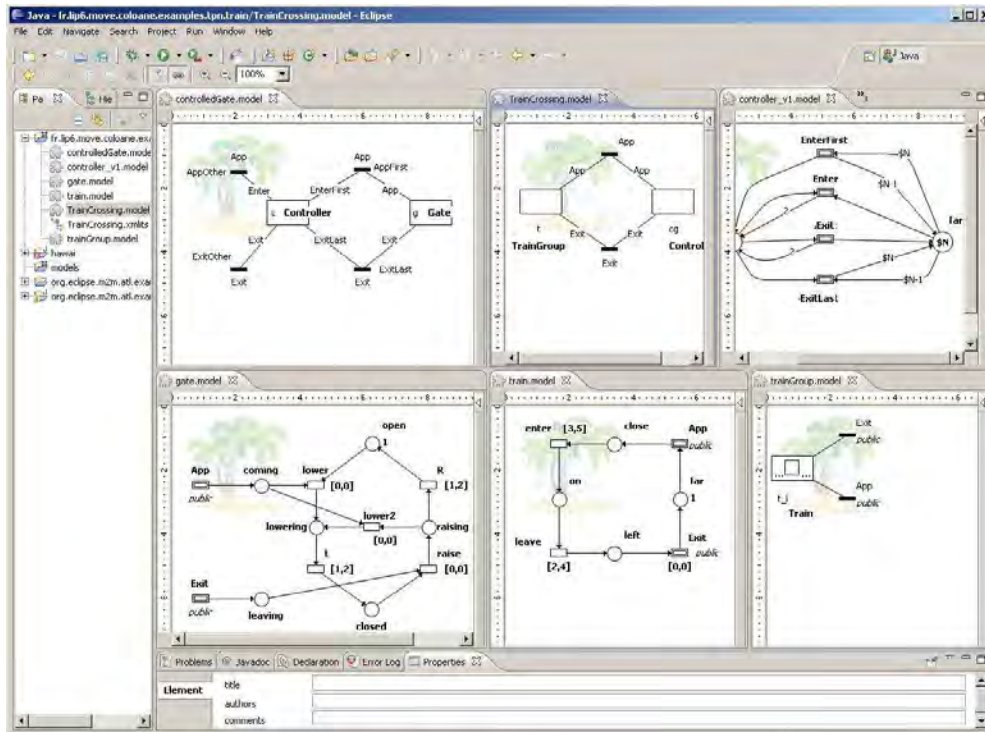


Fig. 9. Coloane based ITS modeler

given process to fire **Myturn**, we need to first reset all *go* places of the processes, and synchronously **Reset** the state of the resource.

In view of these performances, we now compare Roméo/SDD with existing tools.

Dense Time, Explicit data structures. The standard approach for verification of timed models relies on difference bounded matrices (DBMs), an efficient data structure to represent time zones under dense time hypothesis. Its efficiency has led to the development of several verification tools (so-called timed model-checkers) such as TINA [4], Roméo [11] for Time Petri Nets and UPPAAL [17], Kronos [26] for Timed Automata.

Roméo suffers from the discrete state space explosion (*i.e.* in number of classes). It was stopped after one day of CPU and consumed a high amount of memory. The performances of UPPAAL without symmetries (not reported due to lack of space in the paper) are similar to those of Roméo's. Underlying technology is DBM in both cases.

Thanks to the symmetry management, UPPAAL/sym copes very well with these regular models. It uses a canonization procedure which is costly in time, but can in favorable cases represent only a fraction of the state space. It significantly outperforms all the other tools tested except our own tool. However, computation time grows quicker than memory consumption (possibly due to the canonization complexity) and becomes the limiting factor.

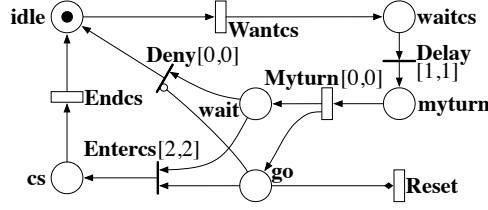


Fig. 10. Process type for Fischer.

$$\langle \text{Process}, N, s_0, \left\{ \begin{array}{l} \langle \text{Wantcs}, ANY, public \rangle, \\ \langle \text{Myturn}, ANY, public \rangle, \\ \langle \text{Endcs}, ANY, public \rangle, \\ \langle \text{Reset}, ALL, public \rangle, \\ \langle \mathbb{1}, ALL, public \rangle \end{array} \right\}, 0 \rangle.$$

Fig. 12. A Scalar set type ProcessGroup

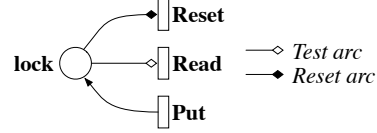


Fig. 11. The resource type.

r : Resource	pg : ProcessGroup	λ
Read	Wantcs	\top
Put	Endcs	\top
Reset	Reset·Myturn	\top
$\mathbb{1}$	$\mathbb{1}$	$\mathbb{1}$

Fig. 13. Synchronization for the complete Fischer system.

Dense Time, Symbolic data structures. To bring the benefits of BDD technology to model-checking of TA, many fully symbolic encodings that use dedicated BDD-like data structures have been proposed (e.g. DDD [20]). The most successful seems to be Clock Restriction Diagrams used in the tool RED [25]. In some instances, this approach can outperform DBM technology, particularly when the number of clocks increases, and backward reachability is used. Other approaches that map the timed reachability problem to a problem solvable using standard BDD exist (e.g. TMV tool [23]), but the performances as reported are comparable to those using DBM.

We compare in this category to the tool RED, which builds a zone graph using Clock Restriction Diagrams in a fully symbolic approach. The fully symbolic approach implemented in RED is fast in CPU time, but also grows very fast in memory. This is consistent with reports from experiments with RED [25], which manages to go a bit further than DBM as it is more resistant to an increase in the number of locations.

Discrete Time, Symbolic data structures. When considering discrete time semantics, the only viable way to tackle the combinatorial explosion seems to be symbolic data structures. Direct encoding of counters with an explicit model-checker is bound to fail (unless some abstraction or acceleration is used), the number of states grows very fast (up to 10^{512} in our experiments). This excludes the use of non symbolic discrete approaches, which could be experimented by using UPPAAL with discrete variables instead of clocks.

Among the different tools based on standard BDD structures and discrete time semantics, SMI (based on Kronos) [7], and Rabbit [6] are the closest to our approach with Roméo/SDD, which also belongs to this category. Although this discretized approach is sensitive to the maximum clock values in a model, it can often outperform dense time approaches.

However, comparison is difficult because both SMI and Rabbit are old prototypes no longer maintained (current distribution of Rabbit is from 2002). The input of Rabbit

Fischer (N is the number of processes)											
N	Roméo			RED		UPPAAL/sym			Roméo/SDD		
	tm	mm	sm	tm	mm	tm	mm	sm	tm	mm	sm
8	1 051	282 108	740 633	11	278 028	0.01	160	137	0.1	2 020	$1.17 \cdot 10^6$
9	73 071	$1.77 \cdot 10^6$	$3.72 \cdot 10^6$	67	785 108	0.03	160	172	0.1	2 156	$6.20 \cdot 10^6$
10	DNF	-	-	652	$2.35 \cdot 10^6$	0.1	160	211	0.1	2 332	$3.26 \cdot 10^7$
170	-	-	-	-	OOM	7 783	47 956	57 971	23	101 896	$2.27 \cdot 10^{120}$
700	-	-	-	-	-	DNF	-	-	1391	$1.82 \cdot 10^6$	$2.66 \cdot 10^{491}$
730	-	-	-	-	-	-	-	-	1803	$2.33 \cdot 10^6$	$2.58 \cdot 10^{512}$

Train (N is the number of trains)											
N	Roméo			RED		UPPAAL/sym			Roméo/SDD		
	tm	mm	sm	tm	mm	tm	mm	sm	tm	mm	sm
6	43.1	36 948	29 640	7	202 412	0.14	908	432	1.5	7 360	$4.83 \cdot 10^6$
7	6 115	377 452	131 517	66	723 428	0.23	3 200	957	2.5	10 304	$6.28 \cdot 10^7$
8	DNF	-	-	-	OOM	1	3 336	2 078	4	14 188	$8.16 \cdot 10^8$
13	-	-	-	-	-	2 634	13 188	79 598	26	56 660	$3.02 \cdot 10^{14}$
15	-	-	-	-	-	60 860	61 256	-	42	86 360	$5.11 \cdot 10^{16}$
16	-	-	-	-	-	DNF	-	-	52	104 848	$6.65 \cdot 10^{17}$
44	-	-	-	-	-	-	-	-	1143	$2.13 \cdot 10^6$	$1.03 \cdot 10^{49}$

Table 1. Performances measured for the *Fischer* and *train* models. Execution time is in seconds (column tm), memory occupation in KB (column mm). Column sm provides a measure of the state space size. DNF means that the computation did not finish within one day, while OOM means computation exceeds 2.4GB memory.

is a variant on a classical product of TA, but without the hierarchical characteristics of ITS. Rabbit measures are not reported in the table because we were unable to operate the tool on the Train model. The Fischer model which is part of Rabbit distribution was managed up to 128 processes in 1587 seconds with 842 MB of memory, which is a good result. We could not experiment further with this tool because it does not allow the use of more than 880MB of memory.

Impact of Scalar set. Roméo/SDD relies on similar principles as Rabbit: discrete time and a fully symbolic representation. The combinatorial explosion due to discrete time is balanced by the efficiency of SDD encoding: over 10^{500} states can be represented.

Roméo/SDD is able to handle models up to much higher parameter values than the other tools. This is due to the use of ITS/SDD that provide: (i) automatic saturation, (ii) shared representation of subsystems and (iii) the cartesian product style definition of the transition relation (involving composition of sums). Various strategies (see [24]) can be used to encode regular models as ITS, in a way that allows to exploit the same kind of symmetries as UPPAAL/sym. A strategy can be configured to encode the TrainGroup or ProcessGroup composite types (which are both scalar sets) with varying width and levels of depth in the hierarchy. We experimented with the standard flat setting (fixed depth of 1 for scalar set) with n instances side by side for the train model (*i.e.* n groups of size 1). Other settings with less groups of higher cardinality (or more generally with depth superior to 1) lead to lower performances. In fact, the final representation can be smaller, but due to a peak effect, increasing the depth does not allow us to solve larger models. This peak effect could result from the strong synchronization due to the delay transition.

For the Fischer model, we used a setting with a recursive definition of hierarchy in the system bounded by 12 variables per level. In other words, the depth is left unbounded while the width is at most 12. The standard flat setting (depth = 1) was able to reach 200 processes while a setting with groups of about 10 processes let us reach 400 processes (depth fixed at 2, 40 groups of ten process).

Although experimentation on industrial case studies remains to be done, this benchmark shows that appropriate data structures applied to discrete time can deal with models that could not be analyzed before due to combinatorial explosion in time and/or memory.

5 Conclusion

This work proposes an approach to compute the state space for a large number of discrete timed components, which relies on hierarchical set decision diagrams (SDD) and scalar sets defined in the ITS framework. It allows a hierarchical definition of a timed system, and offers an efficient fully symbolic reachability engine thanks to the automatic activation of saturation. Because of its generality, ITS can be reused to encode any formalism with discrete time semantics and mix several timed models within the same specification while enabling efficient state space generation.

Performance comparisons with reference tools, for both discrete and dense time, show gains of several orders of magnitude. This significant improvement is due to our fully symbolic encoding, instead of a classical symbolic approach for time zones only.

SDD and ITS are both available as C++ libraries under the terms of GNU LGPL, at <http://ddd.lip6.fr>. A new version of Roméo integrating a SDD engine will soon be available. A front-end to build TPN and their composition is already provided within the Coloane modeling environment. We are currently working at providing full discrete temporal logic model-checking capabilities on top of this reachability computation, with a focus on TCTL.

References

1. R. Alur and D. L. Dill. A theory of timed automata. *Theor. Comp. Sci.*, 126:183–235, 1994.
2. H. R. Andersen. Partial model checking. In *In Proc. of LICS'95*, pages 398–407, 1995.
3. A. Arnold. Nivat's processes and their synchronization. *Theor. Comp. Sci.*, 281(1-2):31–36, 2002.
4. B. Berthomieu, P.-O. Ribet, and F. Vernadat. The tool tina – construction of abstract state spaces for petri nets and time petri nets. *International Journal of Production Research*, 42(4), July 2004. <http://www.laas.fr/tina/>.
5. B. Berthomieu and F. Vernadat. State class constructions for branching analysis of time petri nets. In *Tools and Algorithms for the Construction and Analysis of Systems – TACAS*, volume 2619 of *LNCS*, pages 442–457. Springer, 2003.
6. D. Beyer, C. Lewerentz, and A. Noack. Rabbit: A tool for bdd-based verification of real-time systems. In *Computer-Aided Verification – CAV*, volume 2725 of *LNCS*, pages 122–125. Springer, 2003.
7. M. Bozga, O. Maler, A. Pnueli, and S. Yovine. Some progress in the symbolic verification of timed automata. In *Computer Aided Verification – CAV*, pages 179–190. Springer, 1997.

8. J. Burch, E. Clarke, and K. McMillan. Symbolic model checking: 10^{20} states and beyond. *Information and Computation (Issue for LICS90 best papers)*, 98(2):153–181, 1992.
9. J.-M. Couvreur and Y. Thierry-Mieg. Hierarchical Decision Diagrams to Exploit Model Structure. In *Formal Techniques for Networked and Distributed Systems - FORTE*, volume 3731 of *LNCS*, pages 443–457. Springer, 2005.
10. D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 197–212. Springer, 1989.
11. G. Gardey, D. Lime, M. Magnin, and O. H. Roux. Roméo: A tool for analyzing time Petri nets. In *17th International Conference on Computer Aided Verification (CAV'05)*, volume 3576 of *LNCS*. Springer, July 2005. <http://romeo.rts-software.org/>.
12. A. Hamez, Y. Thierry-Mieg, and F. Kordon. Hierarchical Set Decision Diagrams and Automatic Saturation. In *ICATPN 2008*, volume 5062 of *LNCS*, 2008.
13. M. Hendriks, G. Behrmann, K. G. Larsen, P. Niebert, and F. W. Vaandrager. Adding Symmetry Reduction to Uppaal. In *First International Workshop on Formal Modeling and Analysis of Timed Systems (FORMATS'03)*, volume 2791 of *LNCS*, pages 46–59. Springer, 2003.
14. T. A. Henzinger, Z. Manna, and A. Pnueli. What good are digital clocks? In *Int. Coll. Automata, Languages, and Programming – ICALP*, volume 623 of *LNCS*, pages 545–558. Springer, 1992.
15. C. N. Ip and D. L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9:41–75, 1996.
16. F. Laroussinie and K. G. Larsen. Compositional model checking of real time systems. In *CONCUR*, volume 962 of *LNCS*, pages 27–41. Springer, 1995.
17. K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, Oct 1997. <http://www.uppaal.com/>.
18. M. Magnin, D. Lime, and O. H. Roux. Symbolic state space of Stopwatch Petri nets with discrete-time semantics. In *ICATPN 2008*, volume 5062 of *LNCS*, pages 307–326, 2008.
19. P. M. Merlin. *A study of the recoverability of computing systems*. PhD thesis, 1974.
20. J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. Fully Symbolic Model Checking of Timed Systems using Difference Decision Diagrams. In *ENTCS*. Elsevier Science, 1999.
21. P. Niebert and D. Peled. Efficient model checking for ltl with partial order snapshots. *Theor. Comput. Sci.*, 410(42):4180–4189, 2009.
22. L. Popova. Time Petri Nets State Space Reduction using Dynamic Programming. *Journal of Control and Cybernetics*, 35(3):721–748, 2006.
23. S. A. Seshia and R. E. Bryant. Unbounded, Fully Symbolic Model Checking of Timed Automata using Boolean Methods. In *Computer Aided Verification – CAV*, volume 2725 of *LNCS*, pages 154–166. Springer, 2003.
24. Y. Thierry-Mieg, D. Poitrenaud, A. Hamez, and F. Kordon. Hierarchical set decision diagrams and regular models. In *Tools and Algorithms for the Construction and Analysis of Systems – TACAS*, volume 5505 of *LNCS*, pages 1–15. Springer, 2009.
25. F. Wang. Efficient verification of timed automata with BDD-like structures. *Int. Journal on Soft. Tools for Technology Transfer*, 6(1):77–97, 2004.
26. S. Yovine. Kronos: A verification tool for real-time systems. *International Journal of Software Tools for Technology Transfer*, 1(1–2):123–133, Oct 1997.

On the modularity in Petri Nets of Active Resources^{*}

Vladimir A. Bashkin

Yaroslavl State University
Yaroslavl, 150000, Russia
email: bas@uniyar.ac.ru

Abstract. Petri Nets of Active Resources (AR-nets) represent a dual syntax of Petri nets with a single type of nodes (places and transitions are united) and two types of arcs (input and output arcs are separated). In AR-nets the same token may be considered as a passive resource (produced or consumed by agents) and an active agent (producing or consuming resources) at the same time.

It is shown that the homogeneous structure of nodes in AR-nets allows some specific modular modeling and transformation techniques. Properties of net partitions and reachability-equivalent module replacements are studied.

1 Introduction

Nowadays there exist a lot of Petri net modifications introducing different modular and/or hierarchical syntax. In particular, different high-level formalisms appeared to be quite effective and useful in practice [6, 14]. Many authors use algebraic approach to compositions and decompositions [3, 4], or apply some effective algebraic methods of modular verification [9]. Some models even allow a recursion [7, 12].

As a rule, in compositional Petri nets modules may be linked by synchronized transitions [5, 9, 12] or by common interface places [8]. This is a natural consequence of Petri net syntax. Indeed, the structure of a net is explicitly divided into two classes of elements: places and transitions. Places correspond to the passive component of the system (*state* or *resources*), transitions correspond to its active component (*actions* or *events* or *agents*).

However, explicit separation of nodes into places and transitions is not the only way of Petri net definition. There exists a number of equivalent formalisms with a different separation of elements. Some of them use the duality between places and transitions (the importance of studying this duality was mentioned by C.-A. Petri already in [13]).

In [11] K. Lautenbach introduced a notion of dual place/transition nets. In this formalism the transitions are also marked by special tokens called “t-tokens”.

^{*} This research is partially supported by Russian Fund for Basic Research (projects 09-01-00277, 11-01-00737) and Federal Program “Kadry” (project 02.740.11.0207).

The meaning of t-tokens is that they prevent transitions from being enabled. A transition carrying a t-token cannot be enabled by any marking of p-tokens. Place in the net can be enabled and fired in the dual way. Place firing transforms the marking of t-tokens, arcs for place firing are inverted. So the net can be dualized in the obvious way. K.Lautenbach in his work proposed dual P/T nets as a model of system fault propagation.

In [10] M.Köhler and H.Rölke introduced Super-Dual nets for modeling with dynamic refinement of events. In this formalism transitions are also marked by special tokens called “pokens”, but these pokens *enable* transition firings. Places can also fire, but their firing use a special separate set of arcs called “glow relation” in contrast to common “flow relation”. Super-Dual net can be dualized by interchanging places and transitions, tokens and pokens, flow arcs and glow arcs. In [10] it is proven that Super-Dual nets have the same expressive power as ordinary Petri nets.

In both dual P/T nets and Super-Dual nets duality is based on two types of elements of the system — resources and actions (places and transitions). These elements are represented in the net by vertices of a bipartite oriented graph. However, there is another (implicitly) divided set in every Petri net (and in every other bipartite oriented graph) — the set of arcs. It contains arcs of two crucially different types — input arcs from places to transitions remove tokens, output arcs from transitions to places produce tokens. The explicit separation of this notions allowed us to define an “orthogonal” syntax for Petri nets — Nets of Active Resources (AR-nets) [1].

A definition of an AR-net is a dualized definition of a Petri net. The set of arcs is explicitly transformed into two separate sets of *input arcs* and *output arcs*. The sets of transitions and places are united into a single set of *nodes*. Each node may contain *tokens*. A token in the node may fire, consuming some tokens through input arcs and producing some other tokens through output arcs. So a token simulates behaviour of both an active component (an agent) and a passive component (a resource) at the same time. Therefore the formalism is called “nets of active resources”. AR-nets are well-suited for modeling systems with an explicit definition of an agent [2].

In this paper we study the compositional properties of AR-nets. A module is represented as a subnet defined by some subset of nodes. An interface of the module is a set of arcs linking its nodes with an outer subnet. A module may have four types of links: input, output, production and consumption. First two of them represent actions of the module itself, the other two represent actions of its neighbours. Hence syntactically a module with adjacent links can be treated as a node with adjacent arcs. This generalization is quite natural and does not affects the homogeneity of the graph of the net.

It is shown that a number of net properties may be inherited from the properties of modules of particular types. It is proven that any nested decomposition of the net can be transformed into an equivalent agent/resource decomposition. For a flat decomposition these kinds of transformations are applicable depending only on the chromatic number of the module linkage graph.

A problem of module equivalence w.r.t. system reachability is also defined and studied. A most general case of this equivalence is obviously undecidable. For a simple case of replacement a criterion of equivalence is given. It is shown how a module (with some additional requirements) can be replaced by a single node without affecting the global reachability relation.

The paper is organized as follows. In section 2 we give basic definitions and notations for nets of active resources. In section 3 an AR-module is introduced. We study different types of modules, their properties and methods of decompositions. In section 4 two notions of module equivalence are defined and studied. Section 5 contains some conclusions and directions for possible future work.

2 Preliminaries

Let S be a finite set. A *multiset* M over a set S is a mapping $M : S \rightarrow Nat$, where Nat is the set of natural numbers (including zero), i. e. a multiset may contain several copies of the same element.

For two multisets M, M' we write $M \subseteq M'$ iff $\forall s \in S : M(s) \leq M'(s)$ (the inclusion relation). The sum and the union of two multisets M and M' are defined as usual: $\forall s \in S : (M+M')(s) = M(s)+M'(s)$, $M \cup M'(s) = \max(M(s), M'(s))$.

By $\mathcal{M}(S)$ we denote the set of all finite multisets over S .

For a multiset $M \in \mathcal{M}(S)$ and a subset $S' \subseteq S$ denote a *projection* $M[S'] \in \mathcal{M}(S')$ of M onto S' as follows: $\forall s \in S' : M[S'](s) = M(s)$.

Similarly, for a binary relation $R \subseteq \mathcal{M}(S) \times \mathcal{M}(S)$ a *projection* $R[S'] \subseteq \mathcal{M}(S') \times \mathcal{M}(S')$ is defined as follows: $\forall s_1, s_2 \in S' (s_1, s_2) \in R[S'] \Leftrightarrow (s_1, s_2) \in R$.

Definition 1. [1] A net of active resources is a tuple $N = (V, I, O)$, where

- V is a finite set of resource nodes (vertices);
- $I \subseteq \mathcal{M}(V \times V)$ is a consumption relation (input arcs);
- $O \subseteq \mathcal{M}(V \times V)$ is a production relation (output arcs).

In graphic form the nodes are represented by circles, the consumption relation by dotted arrows and the production relation by solid arrows.

A *marked net of active resources* is a pair (N, M_0) where N is an AR-net and $M_0 \in \mathcal{M}(V)$ is its *initial marking*.

As usual, pictorially the marking is denoted by black dots.

For a node $v \in V$ by $I(\bullet, v)$, $O(v, \bullet)$, $I(v, \bullet)$ and $O(\bullet, v)$ denote the multisets of nodes of *preconditions*, *postconditions*, *consumers* and *producers*: $\forall w \in V$

$$\begin{aligned} I(\bullet, v)(w) &=_{\text{def}} I(w, v); & O(v, \bullet)(w) &=_{\text{def}} O(v, w); \\ I(v, \bullet)(w) &=_{\text{def}} I(v, w); & O(\bullet, v)(w) &=_{\text{def}} O(w, v). \end{aligned}$$

Definition 2. A node $v \in V$ is active in a marking M iff

- $M(v) > 0$ (the node v is not empty);
- $I(\bullet, v) \subseteq M$ (there are enough tokens in all its input nodes).

An active node v may fire yielding a new marking M' s.t.

$$M' =_{def} M - I(\bullet, v) + O(v, \bullet) \text{ (denoted } M \xrightarrow{v} M').$$

Some natural notions:

Let $i \in I$ and $i = (v_1, v_2)$. Then the arc i is called an *input* arc for the node v_2 and a *consuming* arc for the node v_1 . A token in the node v_1 may be *consumed* through the arc i , a token in the node v_2 can *consume* through the arc i .

Let $o \in O$ and $o = (v_1, v_2)$. Then the arc o is called an *output* arc for the node v_1 and a *producing* arc for the node v_2 . A token in the node v_1 can *produce* through the arc o , a token in the node v_2 may be *produced* through the arc o .

It is impossible to define consuming output and producing input. The token may be producing, consuming, produced and consumed at the same time (through different incident arcs). It can be even self-copied (by producing loop) or self-consumed (by consuming loop).

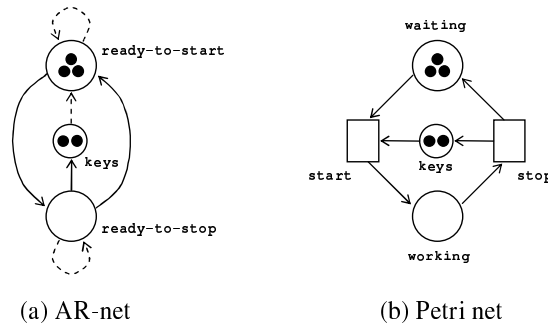


Fig. 1. A model of mutex semaphore with three processes and two resources (AR-model and an equivalent Petri net model).

An example of an AR-net is given on Fig. 1(a). Here we model a system, containing three processes that request two shared resources. Processes are modeled by tokens in the nodes **ready-to-start** and **ready-to-stop**, access keys – by tokens in the node **keys**. The model guarantees that at most two processes can work with resources at the same time. An example of a sequence of firings is given on Fig. 2 (here r_1 denotes **ready-to-start** and r_2 denotes **ready-to-stop**).

On Fig. 1(b) an equivalent Petri net is also presented. Note the difference between two nets. In AR-net the more simple node structure and the more complex arc structure allowed us to use the same node of the graph as a model for both place and transition. For example, the node **ready-to-start** is a replacement for both place **waiting** and transition **start**. Its tokens produce other tokens (in **ready-to-stop**) and are produced by other tokens (by **ready-to-stop**). They also consume other tokens (from **keys**) and are self-consumed (through the loop).

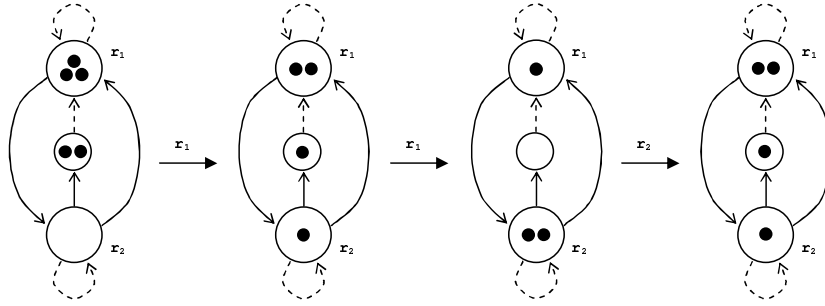


Fig. 2. A sequence of node firings in AR-net.

The notion of firing is extended to sequences in the standard way: for $\sigma \in V^*$ s.t. $\sigma = \sigma'v$ with $v \in V$ we say that $M \xrightarrow{\sigma} M'$ iff $M \xrightarrow{\sigma'} M'' \xrightarrow{v} M'$ for some M'' . A set of reachable markings is defined as follows:

$$\mathcal{R}(N, M_0) =_{\text{def}} \{M \in \mathcal{M}(V) \mid \exists \sigma \in T^* : M_0 \xrightarrow{\sigma} M\}.$$

A node v is *live* in a marked net (N, M_0) iff for any $M \in \mathcal{R}(N, M_0)$ there exists $M' \in \mathcal{R}(N, M)$ such that v is active in M' . A marked net is *live* iff all its nodes are live.

The *reachability relation* is defined as follows:

$$\text{Reach}(N, M_0) =_{\text{def}} \{(M, M') \in \mathcal{M}(V) \times \mathcal{M}(V) \mid \exists \sigma, \sigma' \in T^* : M_0 \xrightarrow{\sigma} M \xrightarrow{\sigma'} M'\}.$$

The syntax of AR-nets substantially differs from the syntax of Petri nets. It may be considered dual: instead of two types of vertices and a single type of arcs we use a single type of vertices and two types of arcs. However, AR-nets define the same class of systems and hence represent yet another variant of Petri net formalism:

Theorem 1. [1] *Nets of active resources are equivalent to Petri nets.*¹

The proof of AR→PN transformation is based on the method, proposed in [10] for Super Dual nets. A node of the AR-net is replaced in the Petri net by a pair (place, transition) (as depicted in Fig. 3). This “flattening” allows to obtain a Petri net with the same reachability relation without any additional loops in the reachability graph.

3 Modular nets

Let $N = (V, I, O)$ be an AR-net. A *module* μ of the net N is defined by some subset of nodes $V_\mu \subseteq V$ (considered as internal nodes of the module).

For a module μ of N denote:

¹ For each AR-net there exists a Petri net with the same reachability and vice versa.

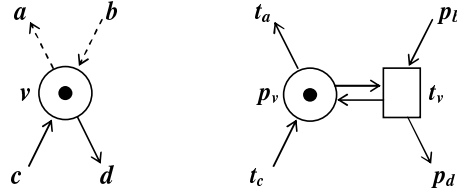


Fig. 3. Transformation of an AR-node into an equivalent Petri net.

- $I_\mu = \{(v, v') \in I \mid v, v' \in V_\mu\}$ – internal input arcs;
- $O_\mu = \{(v, v') \in O \mid v, v' \in V_\mu\}$ – internal output arcs;
- $N_\mu = (V_\mu, I_\mu, O_\mu)$ – a net of the module μ ;
- $A_\mu^i = \{(v, v') \in I \mid v \in (V \setminus V_\mu), v' \in V_\mu\}$ – input links;
- $A_\mu^o = \{(v, v') \in O \mid v \in V_\mu, v' \in (V \setminus V_\mu)\}$ – output links;
- $R_\mu^i = \{(v, v') \in I \mid v \in V_\mu, v' \in (V \setminus V_\mu)\}$ – consuming links;
- $R_\mu^o = \{(v, v') \in O \mid v \in (V \setminus V_\mu), v' \in V_\mu\}$ – producing links.

Informally, A-links represents the observable activity of the module, R-links describe its role as a resource.

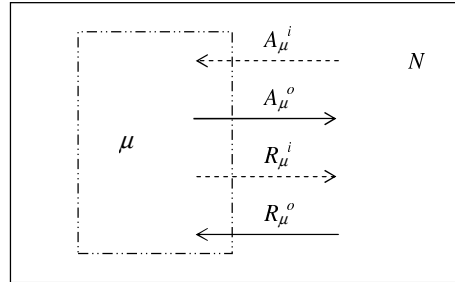


Fig. 4. Four link types in modular AR-nets.

For a marked net (N, M_0) and a module μ a marked net of the module $(N_\mu, (M_0)_\mu)$ is defined straightforwardly: $(M_0)_\mu =_{\text{def}} M_0[V_\mu]$.

Define also a *complement* $\bar{\mu}$ of the module μ as a module, defined by a subset of nodes $V \setminus V_\mu$. A complement of the module may be considered as a *system subnet* of the net.

A well-known model of dining philosophers is given on Fig. 5. For the sake of simplicity we consider only two participants. A module is defined representing the first philosopher. Note that it has only input and output links (elements of A_μ^i and A_μ^o), so the module may be considered as a pure agent.

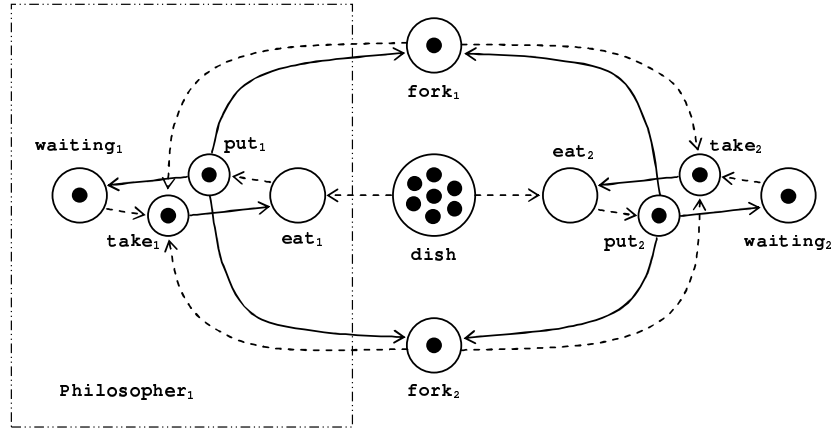


Fig. 5. Two dining philosophers.

A module in an AR-net has almost the same external appearance as a single node: it may consume and produce resources of other modules, and its own resources may be consumed and produced by other modules. Moreover, the relations between modules are naturally denoted by the same constructive elements as at the underlying level of nodes: input and output arcs (links). Hence, the induced hierarchical syntax is quite compact.

Modules having not all four types of links are of a particular interest. We will call a module μ an A-module (resp. R-module) if it has only A-links (resp. R-links). For example, the `philosopher1` is an A-module. Modules with a more restricted interfaces will be denoted using appropriate superscripts: for example, A^iR^o -module has only input and producing links.

Any AR-net may be considered as a composition of modules of different types (Fig. 6). Here are several trivial properties of some of these types:

Proposition 1. *Let (N, M_0) be a marked net and μ be a module of N . Then*

1. $(N_\mu, (M_0)_\mu)$ is unbounded and μ is an A^oR -module $\Rightarrow (N, M_0)$ is unbounded;
2. $(N_\mu, (M_0)_\mu)$ is not live and μ is an AR^i -module $\Rightarrow (N, M_0)$ is not live;
3. $(N_\mu, (M_0)_\mu)$ is live and μ is an A^o -module $\Rightarrow (N_{\bar{\mu}}, (M_0)_{\bar{\mu}})$ is unbounded.

Proof. 1. Since there are no input links, the behavior of the active nodes of the module does not depend on the marking of the system part of the net. Therefore we can take an unbounded run of the module as an unbounded run of the whole net.

2. The AR^i -module cannot obtain any additional tokens from the outside (there are no producing links). So its nodes are not live in the whole net too.
3. Obviously, the live module with only output external links sends the unbounded number of tokens to the outside.

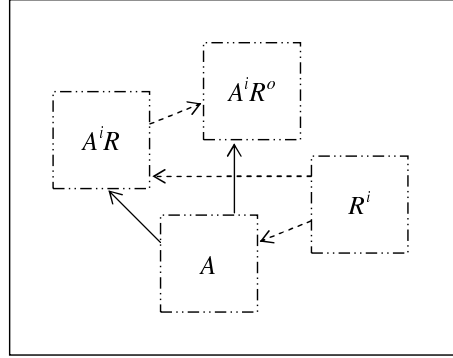


Fig. 6. Links define the role of the module in the system.

Among all 11 possible types of modules pure A- and R-modules (*agents* and *resources*) are the most important. Any interface between two modules may be transformed into an equivalent² A/R interface with one module being an agent, and the other being a resource. Consider a simple procedure transforming input and output links into producing and consuming links:

Lemma 1. *Let (N, M_0) be a marked AR-net and $v \in V$ be a node such that $I(\bullet, v) \neq \emptyset$ or $O(v, \bullet) \neq \emptyset$ (v is active: it can consume or produce tokens).*

Let N' be a net, constructed from N by removing all arcs, participating in $I(\bullet, v)$ and $O(v, \bullet)$ (v became passive), and adding a new node v_t with $I(v_t, \bullet) = O(\bullet, v_t) = \emptyset$ (v_t cannot be produced or consumed), $I(\bullet, v_t) = I(\bullet, v) \cup \{(v, v_t)\}$, $O(v_t, \bullet) = O(v, \bullet) \cup \{(v_t, v)\}$ (v_t simulate in N' the firing of v in N).

Let M'_0 be a marking of N' such that $M'_0[V] = M_0$, $M'_0(v_t) = 1$. Then

$Reach(N, M_0) = Reach(N', M'_0) \cap (V \times V)$ and $\forall M' \in \mathcal{R}(N', M'_0) \quad M'(v_t) = 1$.

Proof. The illustration of such a transformation is given on Fig. 7.

The proof is straightforward – all possible links of a node are considered. Actually, we just separated active and passive properties of node v (just like in Fig. 3). The new node v_t is a *transition*: it behaves completely like an ordinary Petri net transition. Similarly, the node v in N' is a Petri net *place*.

The restructuring, described in Lemma 1, extends the set of nodes by a transition, *always* marked by a single token. So we do not take this node into account when considering the reachability set of the new net.

Corollary 1. 1) *Any module of an AR-net may be transformed into R-module without changing the reachability set of the net;*
 2) *Any module of an AR-net may be transformed into A-module without changing the reachability set of the net.*

² (w.r.t. reachability)

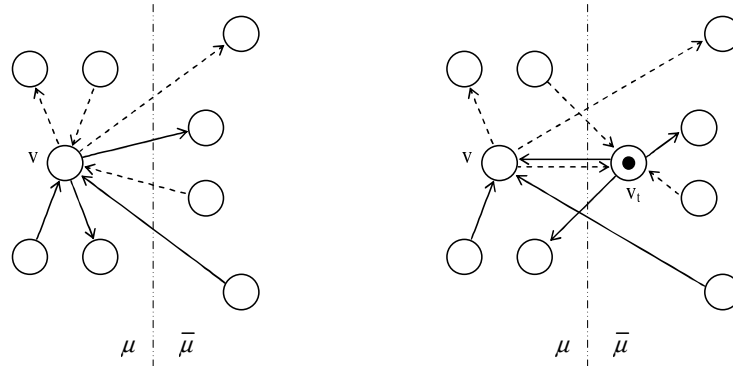


Fig. 7. Transformation of a node into a pair (place,transition).

Proof. (1) Any active node v of the module, having an external link, is replaced by a pair (v, v_t) of a place and a transition. The new transitions are put outside of the module (Fig. 7).

(2) A dual transformation: active nodes of the system part are transformed, transitions are put into the module.

So the interface of any module can be simplified to R-interface or A-interface. Of course, doing this we change the structure of the net (v_t is added). However, this modification is local and does not affect the “inner” part of the module.

In practice A- and R-modules may be considered as “active” and “passive” parts of the system (“control” and “data”). Corollary 1 states that the separation is “relative”: we can easily modify an agent to be a resource and vice versa. This duality is quite trivial in modular AR-nets.

Definition 3. A flat modularization Ω of a net N is a partition of V into non-intersecting modules $\{\mu_1, \dots, \mu_n\}$.

A flat modularization is called a flat A/R-modularization iff every module is either A-module or R-module.

Corollary 2. Let $\Omega = \{\mu_1, \dots, \mu_n\}$ be a flat modularization of N . Let G be a graph with vertices from Ω , such that two vertices μ_i and μ_j are connected in G iff there is an arc between modules μ_i and μ_j in N .

The net N may be transformed into an equivalent (w.r.t. reachability) net N' such that Ω is an A/R-modularization of N' iff the chromatic number of G is 2.

Proof. Modules of one color are transformed into A-modules, of another – into R-modules.

Corollary 2 states that any two-color partition of the net can be transformed to an active/passive partition. Hence we can easily identify control and data structures, corresponding to the given partition: control modules sharing the

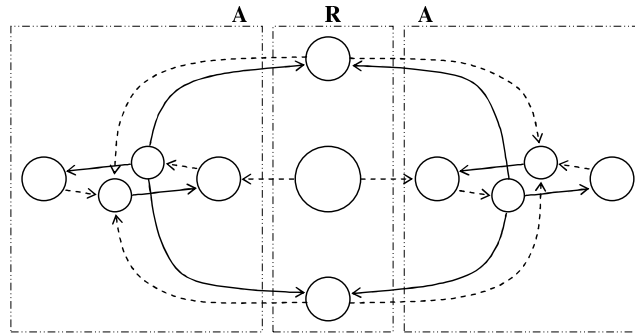


Fig. 8. Flat A/R-modularization.

same data, data modules sharing the same control, chains of linked modules etc. Moreover, control and data subnets are dualizable (Corollary 1).

Definition 4. A nested modularization Ω of a net N is a partition of V into a module μ and a system part $\bar{\mu}$, where μ may also be modularized.

A nested modularization is called a nested A/R-modularization / A-modularization / R-modularization iff every module is either A- or R-module / A-module / R-module.

Corollary 3. For any nested modularization Ω of N this net may be transformed into an equivalent (w.r.t. reachability) net N' such that Ω is a nested α -modularization of N' ($\alpha \in \{A/R, A, R\}$).

Proof. Straightforward. The transformation (if required) must be started from the innermost module.

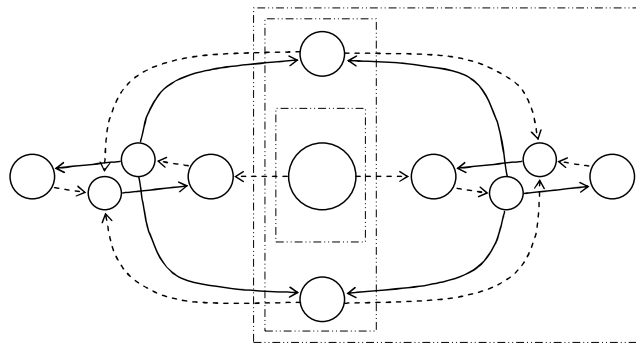


Fig. 9. Nested R-modularization.

The nested modularization allows us to construct a hierarchical structure with any kind of inter-level communication. The combinations of flat and nested modularizations are also possible.

A/R-modularizations are interesting because they allow to incorporate a natural hierarchy into the set of modules. Such hierarchy may have many applications in extended formalisms. For example, active modules may be considered responsible for intermodular data transfer. Another example is security: the R-module is able to hide the exact moments of its transition firings from the agent (the A-module), because agent observes only the already changed state of the resource.

4 Modular reachability

In this section we consider equivalent modules. The key problem is to check whether a particular module can be replaced by another one without harming the behaviour of the whole system. We study the equality of reachability relations of two nets – a fundamental behavioural equivalence, which is stronger than language equivalence and bisimulation.

Definition 5. Consider AR-nets N_1 and N_2 and modules μ_1 and μ_2 of N_1 and N_2 respectively, such that $(N_1)_{\bar{\mu}_1} = (N_2)_{\bar{\mu}_2} = N_{sys}$ for some AR-net $N_{sys} = (V_{sys}, I_{sys}, O_{sys})$ (a same system net). Consider markings M_1, M_2 and M_{sys} of μ_1, μ_2 and N_{sys} respectively.

Marked modules (μ_1, M_1) and (μ_2, M_2) are called equivalent w.r.t. system reachability for a marked system net (N_{sys}, M_{sys}) (SR-equivalent for short) iff

$$\mathcal{Reach}(N_1, M_1 + M_{sys})[V_{sys}] = \mathcal{Reach}(N_2, M_2 + M_{sys})[V_{sys}].$$

Informally, two SR-equivalent modules have the same effect on the system part of the net. They can be replaced by each other without harming the system's reachability set.

An example is given on Fig. 10. The module `Philosopher'` is SR-equivalent to the module `Philosopher1`, shown on Fig. 5. Note that these modules are different: `Philosopher'` has additional state `ready`, in which it has both forks but is not eating.

Theorem 2. SR-equivalence is undecidable for general AR-nets.

Proof. Follows from the undecidability of R-equivalence for general Petri nets (a problem of deciding whether two nets have the same reachability set). Indeed, one can put all “agent nodes” (“transitions”) of compared nets into corresponding modules (and all “resource nodes” aka “places” into system nets) and try to check their SR-equivalence.

Note that in the proof we used active modules (A-modules). Hence SR-equivalence is undecidable even for A-modules. It may be interesting to study

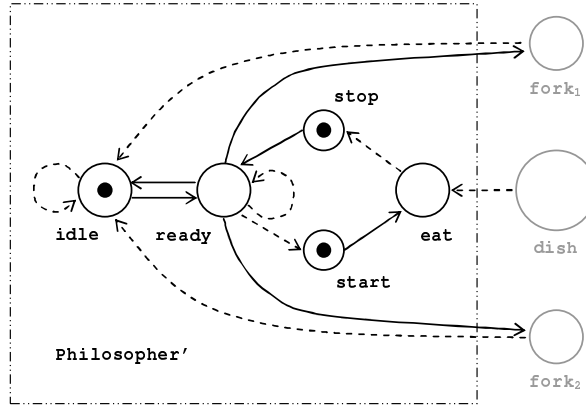


Fig. 10. An SR-equivalent philosopher.

the equivalence for other specific types of modules, in particular, for R-modules. We believe that it is undecidable as well.

Consider a more restricted case of a module. Let both modules have the same interface, i.e. the same set of links and adjacent nodes in the system net: $\forall v \in V_{sys}$

$$\begin{aligned} \sum_{v_1 \in V_{\mu_1}} A_{\mu_1}^o(v_1, v) &= \sum_{v_2 \in V_{\mu_2}} A_{\mu_2}^o(v_2, v); & \sum_{v_1 \in V_{\mu_1}} R_{\mu_1}^i(v_1, v) &= \sum_{v_2 \in V_{\mu_2}} R_{\mu_2}^i(v_2, v); \\ \sum_{v_1 \in V_{\mu_1}} A_{\mu_1}^i(v, v_1) &= \sum_{v_2 \in V_{\mu_2}} A_{\mu_2}^i(v, v_2); & \sum_{v_1 \in V_{\mu_1}} R_{\mu_1}^o(v, v_1) &= \sum_{v_2 \in V_{\mu_2}} R_{\mu_2}^o(v, v_2). \end{aligned}$$

We will say that a module is *compatible* with a system net (and vice versa) iff the net contains nodes required by all links of the module.

Definition 6. *Marked modules (μ_1, M_1) and (μ_2, M_2) , having the same interface, are called universally equivalent w.r.t. system reachability (USR-equivalent for short) iff they are SR-equivalent for any compatible marked system net.*

The USR-equivalence of two modules means that they produce equal sets of markings on passive interface nodes (by active agent links) and obey the same sets of restrictions and commands, coming from active interface nodes (by passive resource links).

A USR-equivalence is a restriction of an SR-equivalence. However, we believe that it is also undecidable.

Consider one of the simplest nontrivial module replacement – let the first module (denoted by μ) be a general AR-net (with some restrictions) and the second one (denoted by ν) be a single node.

Theorem 3. *Let (μ, M) be a marked module s.t.*

1. All active interface nodes of module μ have the same multisets of active links:

$$\exists A^i, A^o \in \mathcal{M}(V_{sys}) \quad \forall v \in V_\mu \\ (A_\mu^i(\bullet, v) = A_\mu^o(v, \bullet) = \emptyset) \vee (A_\mu^i(\bullet, v) = A^i \wedge A_\mu^o(v, \bullet) = A^o).$$

2. All active interface nodes of system net perform operations on the whole numbers of the same multiset of internal nodes:

$$\exists R^{io} \in \mathcal{M}(V_\mu) \quad \forall v \in V_{sys} \quad \exists k^i(v), k^o(v) \in Nat \\ (R_\mu^i(\bullet, v) = k^i(v) \times R^{io} \wedge R_\mu^o(v, \bullet) = k^o(v) \times R^{io}).$$

3. All active internal nodes of the module, affecting nodes of R^{io} , perform the same operation on the whole numbers of R^{io} 's:

$$\exists k^c, k^p \in Nat \quad \forall v \in V_\mu \quad (A_\mu^i(\bullet, v) \cap R^{io} \neq \emptyset \vee A_\mu^o(v, \bullet) \cap R^{io} \neq \emptyset) \Rightarrow \\ (\exists X, Y \in \mathcal{M}(V_\mu) \quad (X \cap R^{io} = Y \cap R^{io} = \emptyset) \wedge \\ (A_\mu^i(\bullet, v) = k^c \times R^{io} + X) \wedge (A_\mu^o(v, \bullet) = k^p \times R^{io} + Y)).$$

4. The initial marking M may be decomposed as $M = M' + m \times R^{io}$, where $m \in Nat$ and $M' \cap R^{io} = \emptyset$.
5. The marked internal net (N_μ, \overline{M}) is live, where \overline{M} denotes a marking, produced from M by emptying all passive interface nodes of μ :

$$\forall v \in V_\mu \quad \overline{M}(v) = \text{def} \begin{cases} 0 & \text{if } (A_\mu^i(\bullet, v) \cup A_\mu^o(v, \bullet)) \neq \emptyset; \\ M(v) & \text{otherwise.} \end{cases}$$

Then (μ, M) is USR-equivalent to a marked single-node module (ν, M_ν) , where

- $V_\nu = \{w\}$;
- $I_\nu(w, w) = k^c$; $O_\nu(w, w) = k^p$;
- $A_\nu^i(\bullet, w) = A^i$; $A_\nu^o(w, \bullet) = A^o$;
- $\forall v \in V_{sys} \quad R_\nu^i(w, v) = k^i(v)$, $R_\nu^o(v, w) = k^o(v)$;
- $M_\nu(w) = m$.

Proof. The proof is technical. It is based on the fact that the liveness of the “unmarked” module implies the liveness of any node of the module in any bigger initial marking (such a marking may be obtained by input from the system). The system does not depend on the actual behaviour of the internal net (note that the effect of all interface firings is the same). It is enough to know that any firing is eventually possible.

An example of correct replacement is given on Fig. 11.

Here the module **Procedure** (live and unbounded) models some computation, that may be performed by one of three computers, initially positioned in the node **idle**. The service **loader** starts the computation, loading one of the computers with an input data. The computer performs calculations (may be, infinitely) and sometimes produces the results (to **output**). It also may be unloaded by another service **unloader**.

The external behaviour of module **Procedure** is relatively simple, so it can be replaced by a single node.

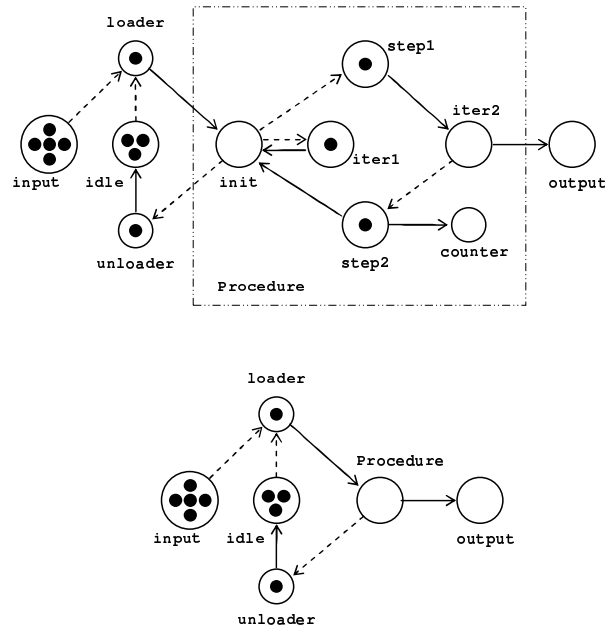


Fig. 11. A module, replaced by a USR-equivalent node.

5 Conclusion and Future work

We tried to identify the distinctive features of AR-nets, that would possibly allow them to be a successful base of some modular (and/or hierarchical) formalism. We also performed some simple analyses of expressiveness of specific modular constructs and decidability of basic module equivalences. It is also shown that modular AR-nets may be a convenient modeling tool.

Since AR-nets are expressively equivalent to general Petri nets, all the results, mentioned in the paper, can be applied to a standard Petri net syntax (in terms of places and transitions). However, in contrast to ordinary Petri nets, the set of nodes is homogeneous here and hence the syntax of module seems quite compact and natural.

We also believe that our work shows the opportunities provided by “coloured” arcs in Petri nets. Two-coloured arcs allowed us to remove the partition of nodes into places and transitions. Obviously, the process of generalization can go further, to a more complex/useful arcs/relations.

The possible directions of a future research in the area of modular AR-nets are: the decidability of certain equivalences of modules; the problem of finding the most effective (smallest/the least connected) decomposition of a given net; the refinement of nodes; the algebraic manipulations with AR-nets and modules; the synchronous compositions (requires additional constructs); etc.

References

1. V.A. Bashkin. Nets of active resources for distributed systems modeling. *Joint Bulletin of NCC&IIS, Comp. Science*. Novosibirsk. 2008. V.28. P.43–54.
2. V.A. Bashkin. Formalization of semantics of systems with unreliable agents by means of nets of active resources. *Programming and Computer Software*, 2010, Vol.36, No.4, P.187–196.
3. E. Best, R. Devillers, M. Koutny. Petri Net Algebra. *EATCS Monographs on TCS*. Springer, Berlin, 2001.
4. E. Best, W. Frączak, R.P. Hopkins, H. Klaudel, E. Pelz. M-nets: an algebra of high level Petri nets, with an application to the semantics of concurrent programming languages. *Acta Inf.*, 1998. Vol.35. P.813–857.
5. S. Christensen, L. Petrucci. Modular analysis of Petri nets. *The Computer Journal*, 2000. Vol.43(3). P.224–242.
6. K. Jensen. Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use. Springer, 1994.
7. S. Haddad, D. Poitrenand. Theoretical aspects of recursive Petri nets. In *Proc. of ATPN'99*. LNCS 1639. Springer, 1999. P.228–247.
8. E. Kindler. A compositional partial order semantics for Petri net components. In *Proc. of ATPN'1997*. LNCS 1248. Springer, 1997. P.235–252.
9. K. Klai, S. Haddad, J.-M. Ilié. Modular Verification of Petri Nets Properties: A Structure-Based Approach. In *Proc. of FORTE'2005*. LNCS 3731. Springer, 2005. P.189–203.
10. M. Köhler, H. Rölke. Super-Dual Nets. In *Proc. of CS&P'2005*. P.271–280.
11. K. Lautenbach. Duality of Marked Place/Transition Nets. Universität Koblenz-Landau, Institut für Informatik, Research Report 18, 2003.
12. I.A. Lomazova. Nested Petri nets - a Formalism for Specification and Verification of Multi-Agent Distributed Systems. *Fundamenta Informaticae*. 2000. V.43. P.195–214.
13. C.-A. Petri. “Forgotten Topics” of Net Theory. In *Proc. of ATPN'1987*. P.500–514.
14. R. Valk. Petri Nets as Token Objects: An Introduction to Elementary Object Nets. LNCS 1420. Springer, 1998. P.1–25.

Optimising the compilation of Petri net models

Lukasz Fronc and Franck Pommereau

IBISC, University of Évry, Tour Évry 2
 523 place des terrasses de l’Agora, 91000 Évry, France
 {fronc,pommereau}@ibisc.univ-evry.fr

Abstract. Compilation of a Petri net model is one way to accelerate its analysis through state space exploration. In this approach, code to explore the Petri net states is generated, which avoids the use of a fixed exploration tool involving an interpretation of the Petri net structure. In this paper, we present a code generation framework for coloured Petri nets targeting various languages (Python, C and LLVM) and featuring optimisations based on peculiarities in models like places types, boundedness, invariants, etc. When adequate modelling tools are used, these properties can be known by construction and we show that exploiting them does not introduce any additional cost while further optimising the generated code. The accelerations resulting from this optimised compilation are then evaluated on various Petri net models, showing speedups and execution times competing with state-of-the-art tools.

Keywords: explicit model-checking, acceleration, model compilation

1 Introduction

System verification through *model-checking* is one of the major research domains in computer science [3]. It consists in defining a formal model of the system to be analysed and then use an automated tool to check whether the expected properties are met or not. In this paper, we consider more particularly the domain of *coloured Petri nets* [18], widely used for modelling, and the *explicit model-checking* approach that enumerates all the reachable states of a model (contrasting with symbolic model-checking that handles directly sets of states).

Among the numerous techniques to speedup explicit model-checking, *model compilation* may be used to generate source code then compiled into machine code to produce a high-performance implementation of the state space exploration. For instance, this approach is successfully used by Helena [5, 24] that generates C code and the same approach is also used by the well-known model-checker Spin [16]. This accelerates computation by avoiding an interpretation of the model that is instead dispatched within a specially generated analyser.

The compilation approach can be further improved by exploiting peculiarities in the model of interest in order to optimise the generated code [19]. For instance, we will see in the paper how 1-boundedness of places may be exploited. Crucially, this information about the model can often be known by construction if adequate modelling techniques are used [27], avoiding any analysis before state

space exploration, which would reduce the overall efficiency of the approach. This differs from other optimisations (that can be also implemented at compile time) like transitions agglomeration implemented in Helena [6].

In this paper, we present a Petri net compiler infrastructure and consider simple optimisations, showing how they can accelerate state space computation. These optimisations are not fundamentally new and similar ideas can be found in Spin for example. However, to the best of our knowledge, this is the first time such optimisations are considered for coloured Petri nets. This allows us for instance to outperform the well-known tool Helena, often regarded as the most efficient explicit model-checker for coloured Petri nets. Moreover, our approach makes use of a flexible high-level programming language, Python, as the colour domain of Petri nets, which enables for quick and easy modelling. By exploiting place types provided in the model, most of Python code in the model can be actually statically typed, allowing to generate efficient machine code to implement it instead of resorting to Python interpretation. This results in a flexible modelling framework that is efficient at the same time, which are in general contradictory objectives. Exploiting a carefully chosen set of languages and technologies, our framework enables the modeller for using an incremental development process based on quick prototyping, profiling and optimisation.

The rest of the paper is organised as follows: we first recall the main notions about coloured Petri nets and show how they can be compiled into a set of algorithms and data structures dedicated to state space exploration. Then section 3 discusses basic optimisations of these elements and section 4 presents benchmarks to evaluate the resulting performances, including a comparison with Helena. For simplicity, we restrict our algorithms to the computation of reachability sets, but they can be easily generalised to compute reachability graphs.

2 Coloured Petri nets and their compilation

A (coloured) Petri net involves a *colour domain* that provides data values, variables, operators, a syntax for expressions, possibly typing rules, etc. Usually, elaborated colour domains are used to ease modelling; in particular, one may consider a functional programming language [18, 29] or the functional fragment (expressions) of an imperative programming language [24, 26]. In this paper we will consider Python as a concrete colour domain. Concrete colour domains can be seen as implementations of a more general *abstract colour domain* providing \mathbb{D} the set of *data values*, \mathbb{V} the set of *variables* and \mathbb{E} the set of *expressions*. Let $e \in \mathbb{E}$, we denote by $\text{vars}(e)$ the set of variables from \mathbb{V} involved in e . Moreover, variables or values may be considered as (simple) expressions, *i.e.*, we assume $\mathbb{D} \cup \mathbb{V} \subseteq \mathbb{E}$. At this abstract level, we do not make any assumption about the typing or syntactical correctness of expressions; instead, we assume that any expression can be evaluated, possibly to $\perp \notin \mathbb{D}$ (undefined value) in case of any error. More precisely, a *binding* is a partial function $\beta : \mathbb{V} \rightarrow \mathbb{D} \cup \{\perp\}$. Then, let $e \in \mathbb{E}$ and β be a binding, we extend the application of β to denote by $\beta(e)$ the evaluation of e under β ; if the domain of β does not include $\text{vars}(e)$ then

$\beta(e) \stackrel{\text{df}}{=} \perp$. The evaluation of an expression under a binding is naturally extended to sets and multisets of expressions.

Definition 1 (Petri nets). A Petri net is a tuple (S, T, ℓ) where S is the finite set of places, T , disjoint from S , is the finite set of transitions, and ℓ is a labelling function such that:

- for all $s \in S$, $\ell(s) \subseteq \mathbb{D}$ is the type of s , i.e., the values that s may contain;
- for all $t \in T$, $\ell(t) \in \mathbb{E}$ is the guard of t , i.e., a condition for its execution;
- for all $(x, y) \in (S \times T) \cup (T \times S)$, $\ell(x, y)$ is a multiset over \mathbb{E} and defines the arc from x toward y .

A marking of a Petri net is a map that associates to each place $s \in S$ a multiset of values from $\ell(s)$. From a marking M , a transition t can be fired using a binding β and yielding a new marking M' , which is denoted by $M[t, \beta]M'$, iff:

- there are enough tokens: for all $s \in S$, $M(s) \geq \beta(\ell(s, t))$;
- the guard is validated: $\beta(\ell(t))$ is true;
- place types are respected: for all $s \in S$, $\beta(\ell(t, s))$ is a multiset over $\ell(s)$;
- M' is M with tokens consumed and produced according to the arcs: for all $s \in S$, $M'(s) = M(s) - \beta(\ell(s, t)) + \beta(\ell(t, s))$.

Such a binding β is called a mode of t at marking M .

For a Petri net node $x \in S \cup T$, we define $\bullet x \stackrel{\text{df}}{=} \{y \in S \cup T \mid \ell(y, x) \neq \emptyset\}$ and $x \bullet \stackrel{\text{df}}{=} \{y \in S \cup T \mid \ell(x, y) \neq \emptyset\}$ where \emptyset is the empty multiset. Finally, we extend the notation vars to a transition by taking the union of the variable sets in its guard and connected arcs.

In the rest of this section and in the next two sections, we consider a fixed Petri net $N \stackrel{\text{df}}{=} (S, T, \ell)$ to be compiled.

2.1 Compilation of coloured Petri nets

In order to allow for translating a Petri net into a library, we need to make further assumptions about its annotations. First, we assume that the considered Petri net is such that, for all transition $t \in T$, and all $s \in S$, $\ell(s, t)$ is either empty or contains a multiset of variables denoted by $X_{s,t} \stackrel{\text{df}}{=} \{x_{s,t,i} \mid 1 \leq i \leq A_{s,t}\}$, where $A_{s,t}$ denotes the arity of the arc from s to t . We also assume that $\text{vars}(t) = \bigcup_{s \in S} \text{vars}(\ell(s, t))$, i.e., all the variables involved in a transition can be bound using input arcs. The second assumption is a classical one that allows to simplify the discovery of modes. The first assumption is made to simplify the presentation: our implementation actually allows for more complex input arcs with pattern matching of structured tokens.

The following definition allows to relate the Petri net to be compiled to the chosen target language, assuming it defines notions of types (statical or dynamical) and functions (with parameters). We need to concretise place types and implement expressions.

Definition 2. A Petri net is compilable to a chosen target language iff:

- for all place $s \in S$, $\ell(s)$ is a type of the target language, interpreted as a subset of \mathbb{D} ;
- for all transition $t \in T$, $\ell(t)$ is a call to a Boolean function whose parameters are the elements of $\text{vars}(t)$;
- for all $s \in t^\bullet$, $\ell(t, s)$ can be evaluated calling a function $f_{t,s}$ whose parameters are the elements of $\text{vars}(t)$ and that returns a multiset over $\ell(s)$, i.e., $f_{t,s}$ is equivalent to a single instruction “**return** $\ell(t, s)$ ”;
- all the functions involved in the annotations terminate.

Given an *initial marking* M_0 , we want to compute the set R of *reachable markings*, i.e., the smallest set such that $M_0 \in R$, and if $M \in R$ and $M[t, \beta]M'$ then $M' \in R$ also. To achieve this computation, we compile the underlying Petri net into a library. The compilation process aims to avoid the use of a Petri net data structure by providing exploration primitives that are specific to the model. These primitives manipulate a unique data structure, *Marking*, that stores a state of the Petri net. The generated library will be used by a client program, a model-checker or a simulator for instance, and used to explore the state space. Thus, it has to respect a fixed API to ensure a correct interfacing with the client program. Moreover, the library relies on primitives (like an implementation of sets) that are assumed to be predefined, as well as code directly taken from the compiled model. This structure is presented in figure 1.

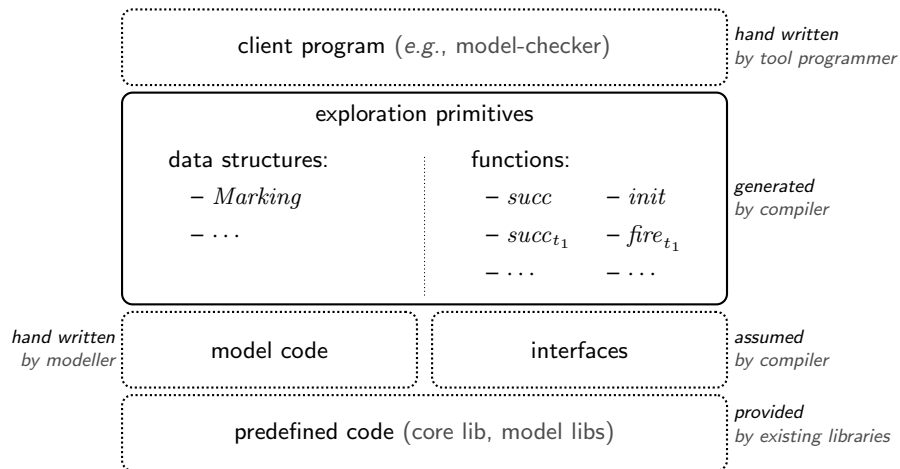


Fig. 1. The compiler generates a library (data structures and functions) that is used by a client program (e.g., a model-checker or a simulator) to explore the state space. This library uses code from the model (i.e., Petri nets annotations) as well as existing data structures (e.g., sets and multisets) forming a core libraries, and accesses them through normalised interfaces. Model code itself may use existing code but it is not expected to do it through any particular interface.

The compiled library is formed of two main parts, data structures which contain the marking structure plus auxiliary structures, and functions for state space exploration. The marking structure is generated following peculiarities of the Petri net in order to produce an efficient data structure. This may include fully generated components or reuse generic ones, the latter have been hand-written and forms a core library that can be reused by the compiler. To make this reusing possible as well as to allow for using alternative implementations of generic components, we have defined a set of interfaces that each generic component implementation has to respect.

A firing function is generated for each transition t to implement the successor relation $M[t, \beta)M'$: given M and the valuation corresponding to β , it computes M' . A successor function $succ_t$ is also generated for each transition t to compute $\{M' \mid M[t, \beta)M'\}$ given a marking M . More precisely, this function searches for all modes at the given marking and produces the set resulting from the corresponding firing function calls. We also produce a function *init* that returns the initial marking of the Petri net, and a global successor function *succ* that computes $\{M' \mid M[t, \beta)M', t \in T\}$ given a marking M , and thus calls all the transition specific successor functions. These algorithms are presented below.

Let $t \in T$ be a transition such that $\bullet t = \{s_1, \dots, s_n\}$ and $t^\bullet = \{s'_1, \dots, s'_m\}$. Then, the transition firing function $fire_t$ can be written as shown in figure 2. This function simply creates a copy M' of M , removes from it the consumed tokens ($x_{s_1, t, 1}, \dots, x_{s_1, t, A_{s_1, t}}, \dots, x_{s_n, t, 1}, \dots, x_{s_n, t, A_{s_n, t}}$) and adds the produced ones before to return M' . One could remark that it avoids a loop over the Petri net places but instead it executes a sequence of statements. Indeed, this is more efficient (no branching penalties, no loop overhead, no array for the functions f_{t, s'_j}, \dots) and the resulting code is simpler to produce. It is important to notice that we do not need a data structure for the modes. Indeed, for each transition t we use a fixed order on $\bullet t$, which allows to implicitly represent a mode through function parameters and avoids data structure allocation and queries.

The algorithm to compute the successors of a marking through a transition enumerates all the combinations of tokens from the input places. If a combination validates the guard then the suitable transition firing function is called and produce a new marking. This is shown in figure 3. The nesting of loops avoids an iteration over $\bullet t$, which saves from querying the Petri net structure and avoids the explicit construction of a binding. Moreover, like f_{t, s'_j} above, g_t is embedded in the generated code instead of being interpreted.

The global successor function *succ* returns the set of all the successors of a marking by calling all transition specific successor functions and accumulating the discovered markings into the same set. This is shown in figure 4.

2.2 Structure of our compilation framework

As shown in figure 5, our compilation framework comprises a frontend part that translates a Petri net into an abstract representation of the target library (including abstracted algorithms and data structures). This representation is then optimised exploiting Petri net peculiarities. For each target language, a

```

firet :  $M, x_{s_1,t,1}, \dots, x_{s_1,t,A_{s_1,t}}, \dots, x_{s_n,t,1}, \dots, x_{s_n,t,A_{s_n,t}} \rightarrow M'$ 


---


 $M' \leftarrow \text{copy}(M)$  // copy marking  $M$ 
 $M'(s_1) \leftarrow M'(s_1) - X_{s_1,t}$  // consume tokens
...
 $M'(s_n) \leftarrow M'(s_n) - X_{s_n,t}$ 
 $M'(s'_1) \leftarrow M'(s'_1) + f_{t,s'_1}(x_{s_1,t,1}, \dots, x_{s_n,t,A_{s_n,t}})$  // produce tokens
...
 $M'(s'_m) \leftarrow M'(s'_m) + f_{t,s'_m}(x_{s_1,t,1}, \dots, x_{s_n,t,A_{s_n,t}})$ 
return  $M'$  // return the successor marking
    
```

Fig. 2. Transition firing algorithm.

```

succt :  $M, next \rightarrow \perp$ 


---


// enumerate every binding of the variables in  $X_{s_n,t}$ 
for  $X_{s_n,t}$  in  $M(s_n)$  do // binds  $x_{s_n,t,1}, \dots, x_{s_n,t,A_{s_n,t}}$ 
...
    for  $X_{s_1,t}$  in  $M(s_1)$  do
        if  $g_t(x_{s_1,t,1}, \dots, x_{s_n,t,A_{s_n,t}})$  then // guard check
             $next \leftarrow next \cup \{fire_t(M, x_{s_1,t,1}, \dots, x_{s_n,t,A_{s_n,t}})\}$  // add a successor marking
        endif
    endfor
...
endfor
    
```

Fig. 3. Transition specific successors computation algorithm.

```

succ :  $M \rightarrow next$ 


---


 $next \leftarrow \emptyset$ 
succt_1( $M, next$ )
...
succt_n( $M, next$ )
return  $next$ 
    
```

Fig. 4. Computation of a marking successors.

dedicated backend translates the abstract representation into code in the target language, and integrate the result with existing components from the core library as well as with the code embedded within the Petri net annotations.

We currently have implemented three backends targeting respectively Python, Cython and LLVM languages. Python is a well-known high-level, dynamically typed, interpreted language [28] that is nowadays widely used for scientific computing [23]. Cython is an extension of Python with types annotations, which allows Cython code to be compiled into efficient C code, thus removing most of the overheads introduced by the Python interpretation [1]. The resulting C code is then compiled to a library that can be loaded as a Python module or from any program in a C-compatible language. The Cython backend is thus also a C

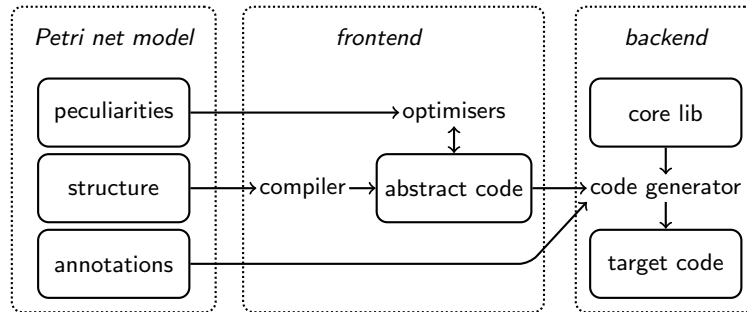


Fig. 5. Structure of the compilation framework.

backend. LLVM is a compiler infrastructure that features a high-level, machine-independent, intermediate representation that can be seen as a typed assembly language [21]. This LLVM code can be executed using a just-in-time compiler or compiled to machine code on every platform supported by the LLVM project.

We consider Petri nets models using Python as their colour domain. The compatibility with the Python and Cython backends is thus straightforward. In order to implement the LLVM backend, we reuse the Cython backend to generate a stripped down version of the target library including only the annotations from the model. This simplified library is then compiled by Cython into C code that can be handled by the LLVM toolchain.

3 Optimisations guided by Petri net structures

3.1 Statically typing a dynamically typed colour domain

This optimisation aims at statically typing the Python code embedded in a Petri net model. In this setting, place types are specified as Python classes among which some are built-in *primitive types* (e.g., *int*, *str*, *bool*, etc.) actually implemented in C. The idea is to use place types to discover the types of variables, choosing the universal type (*object* in Python) when a non-primitive type is found. When all the variables involved in the computation of a Python expression can be typed with primitive types, the Cython compiler produces for it an efficient C implementation, without resorting to the Python interpreter. This results in an efficient pure C implementation of a Python function, similar to the primitive functions already embedded in Python.

In the benchmarks presented in the next section, this optimisation is always turned on. Indeed, without it, the generated code runs at the speed of the Python interpreter, that may dramatically slower, especially when most of data can be statically typed to primitive types (see section 4.3).

3.2 Improving binding discovery

Each function $succ_t$ enumerates the variables from the input arcs in an arbitrary order. The order of the loops thus has no incidence on the algorithm correction,

but it can produce an important speedup as shown in [8]. For instance, consider two places $s_1, s_2 \in \bullet t$. If we know that s_1 is 1-bounded but not s_2 , then it is more efficient to enumerate tokens in s_1 before those in s_2 because the former enumeration is cheaper than the latter and thus we can iterate on s_2 only if necessary. More generally, the optimisation consists in choosing an ordering of the input arcs to enumerate first the tokens from places with a lower boundary or a type with a smaller domain. The optimisation presented in [8] is actually more general and also relies on observations about the order in which variables are bound, which is off-topic in our case considering the restrictions we have imposed on input arcs and guard parameters. However, in the more general setting of our implementation, we are using the full optimisation as described in [8].

In general, place-bounds for an arbitrary Petri net are usually discovered by computing the state space or place invariants [14, 17]. However, using adequate modelling tools or formalisms, this property may be known by construction for many places: in particular, control-flow places in algebras of coloured Petri nets [27] can be guaranteed to be 1-bounded by construction.

3.3 Exploiting 1-bounded places

Let M be a marking and assume a place $s_k \in \bullet t$ that is 1-bounded. In such a case, we can replace the k^{th} “for” loop by an “if” block in the t -specific successor algorithm. Indeed, we know that s_k may contain *at most* one token and so, iterating over $M(s_k)$ is equivalent to check whether s_k is not empty and then retrieve its unique token. This is shown in figure 6, combined with the following optimisation.

3.4 Efficient implementations of place markings

This optimisation consists in replacing a data structure by another one but preserving the interfaces. As a first example, let us consider a 1-bounded place s_k of type $\{\bullet\}$. This is the case for instance for control-flow places in algebras of coloured Petri nets [27]. We assume that $X_{s_k,t} = \{x_{s_k,t,1}\}$ otherwise the transition is dead and can be removed by the compiler. The optimisation consists in replacing the generic data structure for multisets by a much more efficient implementation storing only a Boolean value (*i.e.*, one bit) to indicate whether the place is marked or not.

Similarly, a 1-bounded coloured place may be implemented using a single value to store a token value together with a Boolean to know whether a token is actually present or not. (Another implementation could use a pointer to the token value that would be null whenever the place is empty, but this version suffers from the penalty of dynamic memory management.) An example is given in figure 6 in conjunction with the optimisation from section 3.3.

Finally, a place whose type is *bool* may be implemented as a pair of counters to store the number of occurrences of each Boolean value present in the place. This is likely to be more efficient than a hashtable-based implementation and may be generalised to most types with a small domain.

```

succt : M, next → ⊥


---


... // as in figure 3
for Xsk-1,t in M(sk-1) do
  if M(sk) ≠ ∅ then // tests one bit
    xsk,t,1 ← getsk(M, 1) // directly accesses the unique token
    for Xsk+1,t in M(sk+1) do
      ... // as in figure 3
    endfor
  endif
endfor
... // as in figure 3

```

Fig. 6. An optimisation of the algorithm presented in figure 3 where function $get_{s_k}(M, i)$ returns the i -th token in the domain of $M(s_k)$.

4 Experimental results

The compilation approach presented in this paper is currently being implemented. We use a mixture of different programming languages: LLVM, C, Python and Cython (both to generate C from Python, and as a glue language to interface all the others). More precisely, we use the SNAKES toolkit [25, 26], a library for quickly prototyping Petri net tools, it is used here to import Petri nets and explore their structure. We also use LLVM-Py [22], a Python binding of the LLVM library to write LLVM programs in a “programmatic” way, *i.e.*, avoiding to directly handle source code. It is used here to generate all the LLVM code for state space algorithms. Finally, the core library is implemented using either Python, Cython, LLVM and C. For instance, we directly reuse the efficient sets implementation built into Python, multisets are hand-written in Cython on the top of Python dictionaries (based on hash tables) and a few auxiliary data structures are hand-written directly in C or LLVM. All these language can be mixed smoothly because they are all compatible with C (Python itself is implemented in C and its internal API is fully accessible). The compiler is fully implemented in Python, which is largely efficient enough as shown by our experiments below.

As explained already, the compilation process starts with the front-end that analyses the Petri net and produces an abstract representation (AR) of the algorithms and data structures to be generated. Algorithmic optimisations are performed by the front-end directly on the AR. Then, depending on the selected target language, a dedicated backend is invoked to generate and compile the target code. Further optimisations on data-structure implementation are actually performed during this stage. To integrate these generated code with predefined data structures, additional glue code is generated by the backend. The result is a dynamic library that can be loaded from the Python interpreter as well as called from a C or LLVM program.

The rest of the section presents three case studies allowing to demonstrate the speedups introduced by the optimisations presented above. The machine

used for benchmarks was a standard laptop PC equipped with an Intel i5-520M processor (2.4GHz, 3MB, Dual Core) and 4GB of RAM (2x2GB, 1333MHz DDR3) with virtual memory swapping disabled. We compare our implementation with Helena (version 1.5) and focus on three main aspects of computation: total execution time, model compilation time and state space search time. Each reported measure is expressed in seconds and was obtained as the average of ten independent runs. Finally, in order to ensure that both tools use the same model and compute the same state space, static reduction were disabled in Helena. In order to generate the state space, a trivial client program was produced to systematically explore and store all the successors of all reached markings. All the presented results are obtained using the Cython backend that is presently the most efficient of our three backends, and is as expressive as Python. All the files and programs used for these benchmarks may be obtained at (<http://www.ibisc.fr/~lfronc/SUMo-2011>).

4.1 Dinning Philosophers

The first test case consists in computing the state space of a Petri net model of the *Dinning Philosophers* problem, borrowed from [4]. The considered Petri is a 1-bounded P/T net so we use the corresponding optimisation discussed above. The results for different numbers of philosophers are presented in table 1. We can observe that our implementation is always more efficient than Helena and that the optimisations introduce a notable speedup with improved compilation times (indeed, optimisation actually *simplifies* things). In particular, we would like to stress that the compilation time is very satisfactory, which validates the fact that it is not crucial to optimise the compiler itself.

We also observe that without optimisations our library cannot compute the state space for more than 33 philosophers, and with the optimisations turned on, it can reach up to 36 philosophers. Helena can reach up to 37 philosophers but fails with 38, which can be explained by its use of a state space compression technique that stores about only one out of twenty states [5, 9]. The main conclusion we draw from this example is that our implementation is much faster than Helena on P/T nets, even without optimisations and that compilation times are much shorter. We observe also that it is also faster on bigger state spaces (cases 35 and 36). Following [15], we believe that this is mainly due to avoidance of hash clashes and state comparison when storing states, which validates the efficiency of the model-specific hash functions we generate.

Moreover, with respect to a direct interpretation using SNAKES, our compiler is much faster, for instance, about 900 times faster for 25 philosophers.

4.2 A railroad crossing model

This test case is a model of a railroad crossing system, that generalises the simpler one presented in [27, sec. 3.3] to an arbitrary number of tracks. This system comprises a gate, a set of tracks equipped with green lights, as well as a controller to count trains and command the gates accordingly. For n tracks,

n	states	not optimised			optimised			speedup	Helena			speedup
		t	c	s	t	c	s		t	c	s	
24	103 682	4,7	3,8	0,9	2,4	1,8	0,6	2,0	14,7	12,1	2,6	6,2
25	167 761	5,7	4,0	1,6	3,0	1,8	1,2	1,9	17,5	13,0	4,5	5,8
26	271 443	7,0	4,1	3,0	4,1	1,9	2,2	1,7	20,8	13,0	7,8	5,1
27	439 204	9,6	4,3	5,3	5,9	1,9	4,0	1,6	27,5	14,0	13,5	4,7
28	710 647	14,0	4,8	9,2	9,0	2,0	7,0	1,6	39,1	14,0	25,1	4,3
29	1 149 851	20,9	4,8	16,1	14,9	2,1	12,8	1,4	57,7	15,0	42,7	3,9
30	1 860 498	34,3	5,0	29,3	24,5	2,1	22,4	1,4	92,5	15,0	77,5	3,8
31	3 010 349	56,6	5,4	51,2	41,2	2,1	39,1	1,4	158,0	16,0	142,0	3,8
32	4 870 847	92,8	5,7	87,1	69,8	2,2	67,6	1,3	277,5	16,0	261,5	4,0
33	7 881 196	155,7	6,0	149,8	114,2	2,3	111,9	1,4	437,1	17,0	420,1	3,8
34	12 752 043	m.e.	m.e.	m.e.	193,4	2,3	191,1	m.e.	860,1	17,0	843,1	4,4
35	20 633 239	m.e.	m.e.	m.e.	345,1	2,4	342,7	m.e.	1905,7	18,5	1887,2	5,5
36	33 385 282	m.e.	m.e.	m.e.	598,9	2,4	596,5	m.e.	4253,8	18,5	4235,3	7,1
37	54 018 521	m.e.	m.e.	m.e.	m.e.	m.e.	m.e.	m.e.	10217	19,0	10198,1	m.e.

Table 1. Performance tests based on n philosophers, where t is the total execution time, s is the total state space search time and c is the compilation time. The left-most column entitled “speedup” corresponds to the total time in the unoptimised model divided by the total time in the optimised model. The right-most “speedup” column is the total time spent by Helena divided by the total time in the optimised model. “m.e.” stands for “memory exhausted”.

n	states	not optimised			optimised			speedup	Helena			speedup
		t	c	s	t	c	s		t	c	s	
7	30 626	3.7	3.5	0.2	2.2	2.1	0.2	1.6	14.4	14.0	0.4	6.4
8	124 562	5.1	3.9	1.2	3.1	2.2	0.9	1.6	17.4	15.0	2.4	5.6
9	504 662	10.6	4.3	6.3	6.7	2.4	4.3	1.6	28.9	17.0	11.9	4.3
10	2 038 166	33.9	4.8	29.1	23.8	2.6	21.3	1.4	73.0	18.0	55.0	3.1
11	8 211 530	140.7	5.2	135.5	108.5	2.8	105.7	1.3	345.0	20.0	325.0	3.2
12	33 023 066	m.e.	m.e.	m.e.	m.e.	m.e.	m.e.	m.e.	2721.6	23.0	2698.6	m.e.

Table 2. Performance tests based on n tracks of the railroad model, where t , s , c and “speedup” are as in table 1.

this Petri net has $5n + 11$ black-token 1-bounded places (control flow or flags), 2 integer-typed 1-bounded places (counter of trains and gate position), 3 integer-typed colour-safe places (tracks green lights) and 1 black-token n -bounded place (signals from track to controller). The benchmarks results are shown in table 2.

As above, we notice that our implementation is faster than Helena even without optimisations, and that the optimisations result in similar speedups. We could compute the state space for at most 11 tracks while Helena can reach 12 tracks but fails at 13. As previously, when compared with direct interpretation, we notice that compiled code is much faster: about 350 times faster for 8 tracks (the maximum number SNAKES could handle).

		not optimised			optimised			speedup	exploration speedup
		<i>t</i>	<i>c</i>	<i>s</i>	<i>t</i>	<i>c</i>	<i>s</i>		
with attacker	Python	5.00	0.07	4.93	4.18	0.07	4.11	1.20	1.20
	Cython	5.50	2.36	3.14	4.83	2.02	2.82	1.14	1.11
without attacker	Python	2.58	0.07	2.51	1.85	0.07	1.78	1.40	1.41
	Cython	3.32	2.36	0.97	2.79	2.02	0.77	1.19	1.26

Table 3. Computation of the state space of the security protocol model, where *t*, *s*, *c* and “speedup” columns are as in table 1.

4.3 Security protocol model

The last test case is a model of the Needham-Schroeder public key cryptographic protocol, borrowed from [13]. It embeds about 350 lines of Python code to implement the learning algorithm of a Dolev-Yao attacker. This model comprises 17 places optimised as discussed above whenever possible: 11 are black-token 1-bounded places to implement the control-flow of agents; 6 are coloured 1-bounded places to store the agents’ knowledge; 1 is an unbounded coloured place to store the attacker’s knowledge. Coloured tokens are Python objects or tuples of Python objects. Such a Petri net structure is typical for models of cryptographic protocols like those considered in [13].

For this example, it is not possible to draw a direct comparison with Helena since there is no possible translation of the Python part of the model into the language embedded in Helena. However, compared with a Helena model of the same protocol from [2], we observe equivalent execution times for a SNAKES-based exhaustive simulation and a Helena run. The state space exploration is much faster using Helena but its compilation time is very long (SNAKES does not compile). In our current experiment, we can observe that the compilation time is as good as with other models because the annotations in this model are Python expressions that can be directly copied to the generated code, while Helena needs to translate the annotation of its Petri net model into C code.

The results of the computation of the state space are shown in table 3. We have first presented the whole execution times, then when have excluded the time spent in the Dolev-Yao attacker algorithms that is user-defined external code that no Petri net compiler could optimise. This allows to extract a more relevant speedup that is similar with the previous tests. However, one could observe that the Python version outperforms the Cython version and executes faster and with better speedups. This is actually due to the small size of the state space (only 1234 states), which has two consequences. First, it gives more weight to the compilation time: the Python backend only needs to produce code, whereas the Cython backend also calls Cython and a C compiler. Second, the benefits from Cython can be better observed on the long run because it optimises loops in particular. This is indeed shown in figure 7 that depicts the speedups obtained by compiling to Cython instead of Python with respect to the number of parallel session of the Needham-Schroeder protocol without attacker. To provide an order

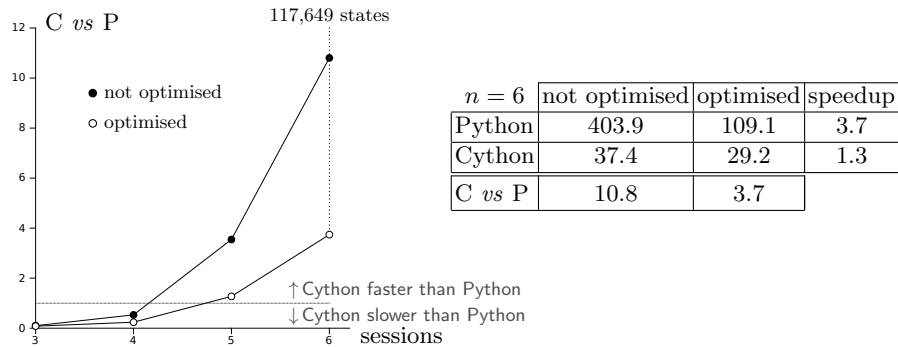


Fig. 7. Speedup of Cython backend compared with Python backend for n sessions of the protocol. Row “C vs P” shows the speedups obtained by compiling to Cython instead of Python, which correspond to the right-most white and black dots on the graph.

of magnitude, we have also shown the number of states for 6 sessions and the speedups obtained from the optimisations. This also allows to observe that our optimisations perform very well on Python also and reduce the execution times, which in turn reduces benefits of using Cython. This is not true in general but holds specially on this model that comprises many Python objects that cannot be translated to C efficient code. So, the Cython code suffers from many context switching between C and Python parts.

We would like to note also that the *modelling time and effort* is much larger when developing a model using the language embedded in Helena rather than using a full-featured language like Python. So, there exists a trade-off between modelling, compilation and verification times that is worth considering. This is why we consider as crucial for our compiler to enable the modeller for quick prototyping with incremental optimisation, as explained in [1]. In our case, the Python implementation of the attacker may be compiled using Cython and optimised by typing critical parts (*i.e.*, main loops). Compared with the implementation of the Dolev-Yao attacker using Helena colour language from [2], the Python implementation we have used is algorithmically better because it could use Python efficient hash-based data structures (sets and dictionaries) while Helena only offers sequential data structures (arrays and lists), which is another argument in favour of using a full-featured colour language.

As a conclusion about performances for this test case, let us sum up interesting facts: SNAKES and Helena versions run in comparable times, the latter spends much more time in compilation but the former has more efficient data structures; our compilation is very efficient; our Python backend is typically 10 times faster than SNAKES simulation; our Cython backend performs as well as the Python backend in this case. So we could reasonably expect very good performances on a direct comparison with Helena, for not too large state spaces.

5 Conclusion

We have shown how a coloured Petri net can be compiled to produce a library that provides primitives to compute the state space. Then, we have shown how different kinds of optimisations can be considered, taking into account peculiarities in the model. We have considered places types and boundaries in particular. Finally, our experiments have demonstrated the relevance of the approach, showing both the benefits of the optimisations (up to almost 2 times faster) and the overall performance with respect to the state-of-the-art tool Helena (up to 7 times faster). With respect to direct interpretation, our compilation approach is up to 900 times faster. Moreover, we have shown that a well chosen mixture of high- and low-level programming languages enables the modeller for quick prototyping with incremental optimisation, which allows to obtain results with reduced time and efforts. Our comparison with Helena also showed that our compilation process is much faster in every case (around 7 times faster). Our optimisations rely on model-specific properties, like place types and boundaries, and do not introduce additional compilation time, instead optimisation may actually simplify things and fasten compilation.

The idea of exploiting models properties has been defended in [27] and successfully applied to the development of a massively parallel state space exploration algorithm for Petri net models of security protocols [13], or in [20] to reduce the state space of models of multi-threaded systems. Let us also remark that the LLVM implementation of the algorithms and code transformations (*i.e.*, optimisations) presented in this paper has been formally proved in [11, 12], which is an important aspect when it comes to perform verification. Crucially, these proofs rely on our careful modular design using fixed and formalised interfaces between components.

Our current work is focused toward finalising our compiler, and then introducing more optimisations. In particular, we would like to improve memory consumption by introducing memory sharing, and to exploit more efficiently the control flow places from models specified using algebras of Petri nets [27]. In parallel, we will develop more case studies to assess the efficiency of our approach. We are also investigating a replacement of the Helena compilation engine with ours, allowing to bring our performances and flexible modelling environment to Helena while taking advantage of its infrastructure, in particular the memory management strategies [7, 9, 10] and the static Petri nets reductions [6].

References

1. S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D.S. Seljebotn, and K. Smith. Cython: The best of both worlds. *Computing in Science & Engineering*, 13(2), 2011.
2. R. Bouroulet, H. Klaudel, and E. Pelz. Modelling and verification of authentication using enhanced net semantics of SPL (Security Protocol Language). In *ACSD'06*. IEEE Computer Society, 2006.
3. E. Clarke, A. Emerson, and J. Sifakis. Model checking: Algorithmic verification and debugging. *ACM Turing Award*, 2007.

4. R. Esser. Dining philosophers Petri net. (<http://goo.gl/j0Fh5>), 1998.
5. S. Evangelista. *Méthodes et outils de vérification pour les réseaux de Petri de haut niveau*. PhD thesis, CNAM, Paris, France, 2006.
6. S. Evangelista, S. Haddad, and J.-F. Pradat-Peyre. Syntactical colored Petri nets reductions. In *ATVA'05*, volume 3707 of *LNCS*. Springer, 2005.
7. S. Evangelista and L.M. Kristensen. Search-order independent state caching. *ToP-NOC III*, to appear, 2010.
8. S. Evangelista and J.-F. Pradat-Peyre. An efficient algorithm for the enabling test of colored Petri nets. In *CPN'04*, number 570 in DAIMI report PB. University of Århus, Denmark, 2004.
9. S. Evangelista and J.-F. Pradat-Peyre. Memory efficient state space storage in explicit software model checking. In *SPIN'05*, volume 3639 of *LNCS*. Springer, 2005.
10. S. Evangelista, M. Westergaard, and L.M. Kristensen. The ComBack method revisited: caching strategies and extension with delayed duplicate detection. *ToPNOC III*, 5800:189–215, 2009.
11. L. Fronc. Analyse efficace des réseaux de Petri par des techniques de compilation. Master's thesis, MPRI, university of Paris 7, 2010.
12. L. Fronc and F. Pommereau. Proving a Petri net model-checker implementation. Technical report, IBISC, 2011. Submitted paper.
13. F. Gava, M. Guedj, and F. Pommereau. A BSP algorithm for the state space construction of security protocols. In *PDMC'10*. IEEE Computer Society, 2010.
14. H.J. Genrich and K. Lautenbach. S-invariance in predicate/transition nets. In *European Workshop on Applications and Theory of Petri Nets*, 1982.
15. G.J. Holzmann. An improved protocol reachability analysis technique. *Software, Practice and Experience*, 18(2), 1988.
16. G.J. Holzmann and al. Spin, formal verification. (<http://spinroot.com>).
17. Kurt J. Coloured Petri nets and the invariant-method. *Theoretical Computer Science*, 14(3), 1981.
18. K. Jensen and L.M. Kristensen. *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*. Springer, 2009, ISBN 978-3-642-00283-0.
19. R. Jourdier. Compilation de réseaux de Petri colorés. Master's thesis, University of Évreux, 2009.
20. H. Klaudel, M. Koutny, E. Pelz, and F. Pommereau. State space reduction for dynamic process creation. *Scientific Annals of Computer Science*, 20, 2010.
21. C. Lattner and al. The LLVM compiler infrastructure. (<http://llvm.org>).
22. R. Mahadevan. Python bindings for LLVM. (<http://www.mdevan.org/llvm-py>).
23. K.J. Millman and M. Aivazis, editors. *Python for Scientists and Engineers*, volume 13(2) of *Computing in Science & Engineering*. IEEE Computer Society, 2011.
24. C. Pajault and S. Evangelista. Helena: a high level net analyzer. (<http://helena.cnam.fr>).
25. F. Pommereau. SNAKES is the net algebra kit for editors and simulators. (<http://www.ibisc.univ-evry.fr/~fpommereau/snakes.htm>).
26. F. Pommereau. Quickly prototyping Petri nets tools with SNAKES. *Petri net newsletter*, 2008.
27. F. Pommereau. *Algebras of coloured Petri nets*. LAMBERT Academic Publishing, October 2010, ISBN 978-3-8433-6113-2.
28. Python Software Foundation. Python programming language. (<http://www.python.org>).
29. C. Reinke. Haskell-coloured Petri nets. In *IFL'99*, volume 1868 of *LNCS*. Springer, 1999.

Generalized Büchi Automata versus Testing Automata for Model Checking

A.-E. Ben Salem^{1,2}, A. Duret-Lutz¹, and F. Kordon²

¹ LRDE, EPITA, Le Kremlin-Bicêtre, France

ala@lrde.epita.fr, adl@lrde.epita.fr

² LIP6, CNRS UMR 7606, Université P. & M. Curie — Paris 6, France

Fabrice.Kordon@lip6.fr

Abstract. Geldenhuys and Hansen have shown that a kind of ω -automaton known as *testing automata* can outperform the Büchi automata traditionally used in the automata-theoretic approach to model checking [8]. This work completes their experiments by including a comparison with generalized Büchi automata; by using larger state spaces derived from Petri nets; and by distinguishing violated formulæ (for which testing automata fare better) from verified formulæ (where testing automata are hindered by their two-pass emptiness check).

1 Introduction

Context The automata-theoretic approach to model checking linear-time properties [23] splits the verification process into four operations:

1. Computation of the state-space for the model M . This state-space can be seen as an ω -automaton A_M whose language, $\mathcal{L}(A_M)$, represent all possible executions of M .
2. Translation of the temporal property φ into a ω -automaton $A_{\neg\varphi}$ whose language, $\mathcal{L}(A_{\neg\varphi})$, is the set of all executions that would invalidate φ .
3. Synchronization of these automata. This constructs a product automaton $A_M \otimes A_{\neg\varphi}$ whose language, $\mathcal{L}(A_M) \cap \mathcal{L}(A_{\neg\varphi})$, is the set of executions of M invalidating φ .
4. Emptiness check of this product. This operation tells whether $A_M \otimes A_{\neg\varphi}$ accepts an infinite word, and can return such a word (a counterexample) if it does. The model M verifies φ iff $\mathcal{L}(A_M \otimes A_{\neg\varphi}) = \emptyset$.

Problem Different kinds of ω -automata have been used with the above approach. In the most common case, a property expressed as an LTL (linear-time temporal logic) formula is converted into a Büchi automaton with state-based acceptance, and a Kripke structure is used to represent the state-space of the model.

In our tools, we prefer to represent properties using *generalized* (i.e., multiple) Büchi acceptance conditions *on transitions* rather than on states [7]. Any algorithm that translates LTL into a Büchi automaton has to deal with generalized Büchi acceptance conditions at some point, and the process of *degeneralizing* the Büchi automaton often increases its size. Several emptiness-check algorithms can deal with generalized Büchi acceptance conditions, making such an a degeneralization unnecessary and even costly [5]. Moving the acceptance conditions from the states to the transitions also reduces the size of the property automaton [3, 10].

Unfortunately, having a smaller property automaton $A_{\neg\varphi}$ does not always imply that the product with the model ($A_M \otimes A_{\neg\varphi}$) will be smaller, and it is the size of this product that really affects the efficiency of the model checking. Instead of targeting smaller property automata, some people have attempted to build automata that are *more deterministic* [21]; however even this does not guarantee the product to be smaller.

Hansen et al. [11] introduced a new kind of ω -automaton called *Testing Automaton*. These automata are less expressive than Büchi automata since are tailored to represent *stuttering-insensitive* properties (such as any LTL property that does not use the X operator). Also they are often a lot larger than their equivalent Büchi automaton, but surprisingly their good determinism often lead to a smaller product. The reasons why and the conditions under which testing automata perform better are still mysterious [8].

Objectives The objective of this paper is to evaluate efficiency of LTL model checking with these three kinds of ω -automata: classical Büchi Automata (BA), Transition-based Generalized Büchi automata (TGBA), and Testing Automata (TA). Our main motivation is to try to establish some rough rules to choose automatically and *a priori* the technique that seems most suitable to check a given *stuttering-insensitive* property on a given model. This is of interest when a tool offers the choice of several techniques, which is the case for our model checker Spot [16].

Contents Section 2 provides a brief summary of the three ω -automaton and pointers to their associated operations for model checking. Then section 3 reports our experimentation procedure and its results before a discussion in section 4.

2 Presentation of the three Approaches

Let AP designate the set of *atomic proposition* of the model that we might want to use to build a linear-time property. Any state of the model can be labeled by a valuation of these atomic propositions. We denote by $K = 2^{AP}$ the set of these valuations. For instance if $AP = \{a, b\}$, then $K = 2^{AP} = \{\bar{a}\bar{b}, \bar{a}b, a\bar{b}, ab\}$. An execution of the model is simply an infinite sequence of such valuations, i.e., an element from K^ω . A property can be seen as a set of sequences, i.e. a subset of K^ω .

This section presents the three kinds of automata we compare in this paper: Transitions-based Generalized Büchi Automata, Büchi Automata and Testing Automata. For all of them, we explain how they recognize subsets of K^ω to show their differences. We do not detail the actual operations that must be performed to model check a system which each approach because this has already been done in other works.

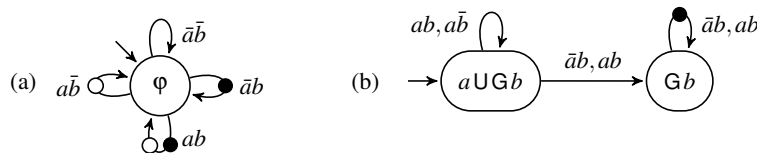


Fig. 1: (a) A TGBA with acceptance conditions $F = \{\bullet, \circ\}$ recognizing the LTL property $\varphi = GFa \wedge GFb$. (b) A TGBA with $F = \{\bullet\}$ recognizing the LTL property $aUGb$.

2.1 Transition-based Generalized Büchi Automata

A Transition-based Generalized Büchi Automata (TGBA) [10] over an alphabet $K = 2^{AP}$ is an ω -automaton where transitions are labeled by letters from K and some acceptance conditions. In our context, the TGBA represents the LTL property to verify.

Definition 1 A TGBA can be formally represented by a tuple $G = \langle S, I, R, F \rangle$ where:

- S is finite set of states,
- $I \subseteq S$ is the set of initial states,
- F is a finite set of acceptance conditions,
- $R \subseteq S \times 2^K \times 2^F \times S$ is the transition relation, where each element (s_i, K_i, F_i, d_i) represents a transition from state s_i to state d_i labeled by the non-empty set of letters K_i , and the set of acceptance conditions F_i .

An execution $w = k_0 k_1 k_2 \dots \in K^\omega$ is accepted by G if there exists an infinite path $(s_0, K_0, F_0, s_1)(s_1, K_1, F_1, s_2)(s_2, K_2, F_2, s_3) \dots \in R^\omega$ where:

- $s_0 \in I$, and $\forall i \in \mathbb{N}, k_i \in K_i \subseteq K$ (the execution is recognized by the path),
- $\forall f \in F, \forall i \in \mathbb{N}, \exists j \geq i, f \in F_j$ (each acceptance condition is visited infinitely often).

Fig. 1 shows two examples of TGBA: one deterministic TGBA derived from the LTL formula $\mathbf{GF}a \wedge \mathbf{GF}b$, and one non-deterministic TGBA derived from $a \mathbf{UG}b$. The LTL formulæ that label states represent the property accepted starting from this state of the automaton: they are shown for the reader's convenience but not used for model checking. As can be inferred from Fig. 1(a), an LTL formula such as $\bigwedge_{i=1}^n \mathbf{GF} p_i$ can be represented by a one-state deterministic TGBA with n acceptance conditions.

Model checking using TGBA When doing model checking with TGBA the two important operations are the translation of the linear-time property ϕ into a TGBA $A_{\neg\phi}$ and the emptiness check of the product $A_M \otimes A_{\neg\phi}$. We know of at least four algorithms that purportedly translate LTL formulæ into TGBA [10, 3, 4, 22]. The one we use is based on Couvreur's LTL translation algorithm [3].

Testing a TGBA for emptiness amounts to the search of a strongly connected component that contains at least one occurrence of each acceptance condition. It can be done in two different way: either with a variation of Tarjan or Dijkstra algorithm [3] or using several nested depth-first searches to save some memory [22]. The latter proved to be slower [5], so we are using Couvreur's SCC-based emptiness check algorithm [3]. Another advantage of the SCC-based algorithm is that their complexity does not depend on the number of acceptance conditions.

2.2 Büchi Automata

A Büchi Automaton (BA) has only one acceptance condition that is state-based.

Definition 2 A BA over the alphabet $K = 2^{AP}$ is a tuple $B = \langle S, I, R, F \rangle$ where:

- S is a set of finite set states,
- $I \subseteq S$ is the set of initial states,
- $F \subseteq S$ is a finite set of acceptance states,
- $R \subseteq S \times 2^K \times S$ is the transition relation where each transition is labeled by a set of letters of K .

An execution $w = k_0k_1k_2\dots \in K^\omega$ is accepted by B if there exists an infinite path $(s_0, K_0, s_1)(s_1, K_1, s_2)(s_2, K_2, s_3)\dots \in R^\omega$ such that:

- $s_0 \in I$, and $\forall i \in \mathbb{N}, k_i \in K_i$ (the execution is recognized by the path),
- $\forall i \in \mathbb{N}, \exists j \geq i, s_j \in F$ (at least one acceptance state is visited infinitely often).

Model checking using BA A BA can be obtained from a TGBA by a procedure known as *degeneralization* [3, 10]. In a worst case, a TGBA with s states and n acceptance conditions will be degeneralized into a BA with $s \times (n + 1)$ states (and one acceptance condition). This is what we do in our experiments. Alternatives include the translation of the property into a *state-based* generalized automaton which can then also be degeneralized, or the translation of the property into an alternating Büchi automaton that is then converted into a BA using the Miyano-Hayashi construction [15].

The emptiness check algorithms that can deal with TGBA will also work on BA (a BA can be seen as a TGBA by pushing the acceptance conditions on the transition leaving acceptance states). But it can also be done using two nested depth-first searches. The comparison of these different emptiness checks has raised many studies [9, 20, 5].

Fig. 2 shows the same properties as Fig. 1, but expressed as Büchi automata. The automaton from Fig. 2(a) was built by degeneralizing the TGBA from Fig. 1(a). The worst case of the degeneralization occurred here, since the TGBA with 1 state and n acceptance conditions was degeneralized into a BA with $n + 1$ states. It is known that no BA with less than $n + 1$ states can recognize the property $\bigwedge_{i=1}^n \text{GF } p_i$ so this Büchi automaton is optimal [2]. The property $a \text{UG} b$, on the other hand, is easier to express: the BA has the same size as the TGBA.

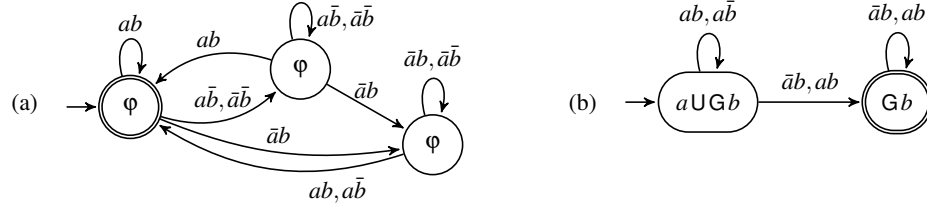


Fig. 2: Two example BA, with acceptance states shown as double circles. (a) A BA for the LTL property $\varphi = \text{GF } a \wedge \text{GF } b$ obtained by degeneralizing the TGBA for Fig. 1(a). (b) A BA for the LTL property $a \text{UG} b$.

2.3 Testing Automata

A property, i.e., a set of infinite sequences $\mathcal{P} \subseteq K^\omega$, is *stuttering-insensitive* iff any sequence $k_0k_1k_2\dots \in \mathcal{P}$ remains in \mathcal{P} after repeating any valuation k_i . In other words, \mathcal{P} is stuttering-insensitive iff

$$k_0k_1k_2\dots \in \mathcal{P} \iff k_0^{i_0}k_1^{i_1}k_2^{i_2}\dots \in \mathcal{P} \text{ for any } i_0 > 0, i_1 > 0 \dots$$

It is well known that any $\text{LTL} \setminus X$ formula (i.e. an LTL formula that does not use the X operator) describes a stuttering-insensitive property. (It is possible to build some stuttering-insensitive LTL formulæ using the X operator [6].)

Testing Automata (TA) were introduced by Hansen et al. [11] to represent stuttering-insensitive properties. While a Büchi automaton observes the value of the atomic propositions AP , the basic idea of TA is to detect the *changes* in these values; if a valuation of AP does not change between two consecutive valuations of an execution, the TA can stay in the same state. To detect execution that ends by stuttering in the same TA state, a new kind of acceptance states is introduced: "livelock acceptance states".

If A and B are two valuations, let us note $A \oplus B$ the symmetric set difference, i.e. the set of atomic propositions that changed. E.g. $a\bar{b} \oplus ab = \{b\}$.

Definition 3 A TA over the alphabet $K = 2^{AP}$ is a tuple $T = \langle S, I, U, R, F, G \rangle$. where:

- S is a finite set of states,
- $I \subseteq S$ is the set of initial states,
- $U : I \rightarrow K$ is a function mapping each initial state to a symbol of K interpreted as a valuation (the initial configuration),
- $R \subseteq S \times K \times S$ is the transition relation where each transition (s, k, d) is labeled by a changeset: $k \in K = 2^{AP}$ is interpreted as a set of atomic propositions that should change between states s and d ,
- $F \subseteq S$ is a set of Büchi acceptance states,
- $G \subseteq S$ is a set of livelock acceptance states.

An execution $w = k_0 k_1 k_2 \dots \in K^\omega$ is accepted by T if there exists an infinite sequence $(s_0, k_0 \oplus k_1, s_1)(s_1, k_1 \oplus k_2, s_2) \dots (s_i, k_i \oplus k_{i+1}, s_{i+1}) \dots \in (S \times K \times S)^\omega$ such that:

- $s_0 \in I$ with $U(s_0) = k_0$,
- $\forall i \in \mathbb{N}$, either $(s_i, k_i \oplus k_{i+1}, s_{i+1}) \in R$ (we are progressing in the testing automaton), or $k_i = k_{i+1} \wedge s_i = s_{i+1}$ (the execution is stuttering and the TA does not progress),
- Either, $\forall i \in \mathbb{N}$, $(\exists j \geq i, k_j \neq k_{j+1}) \wedge (\exists l \geq i, s_l \in F)$ (the automaton is progressing in a Büchi-accepting way), or, $\exists n \in \mathbb{N}$, $(s_n \in G \wedge (\forall i \geq n, s_i = s_n \wedge k_i = k_n))$ (the sequence reaches a livelock acceptance state and then stay on that state because the execution is stuttering).

Construction of a Testing Automaton from a Büchi Automaton From a BA $B = (S_B, I_B, R_B, F_B)$ over the alphabet $K = 2^{AP}$, we obtain a TA $T = (S_T, I_T, U_T, R_T, F_T, G_T)$ representing the same property in two steps [8]:

1. Converting B into an intermediate form of T with $G_T = \emptyset$:
 - $S_T = S_B \times K$, $I_T = I_B \times K$, $F_T = F_B \times K$, and $G_T = \emptyset$
 - $\forall (s, k) \in I_T$, $U_T((s, k)) = k$
 - $\forall (s_1, k_1) \in S_T, \forall (s_2, k_2) \in S_T$,
 $((s_1, k_1), k_1 \oplus k_2, (s_2, k_2)) \in R_T \iff \exists k \in 2^K, ((s_1, k, s_2) \in R_B) \wedge (k_1 \in k)$
2. Filling G_T to simplify T . For that, compute all strongly connected components using only stuttering transitions (i.e., transitions labeled by \emptyset). If such a SCC is not trivial (i.e., it contains a cycle) and contains a Büchi acceptance state, then add all its states to G_T . Add to I_T or G_T any state that can respectively reach I_T or G_T using only stuttering transitions. Finally remove all stuttering transitions from R_T .

Additionally, the TA can be minimized by merging bisimilar states.

Fig. 3 shows the automaton constructed for $aUGb$ by applying the above construction on the automaton from Fig. 2(b). The TA for $GFa \wedge GFb$ is too big to be shown: it has 11 states and 64 transitions.

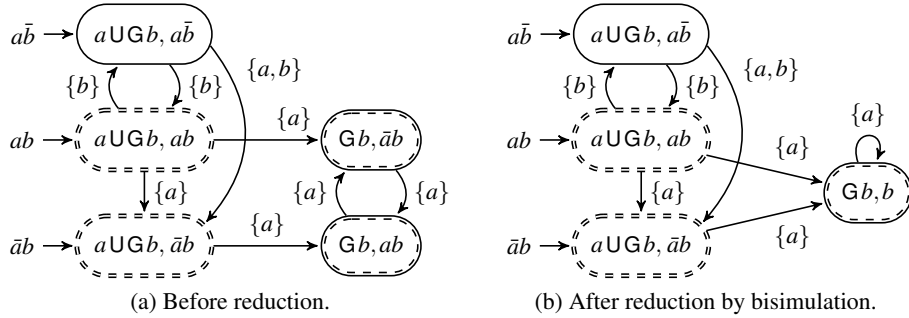


Fig. 3: Two TA for the LTL formula $aUGb$. States with a double enclosure belong to either F or G : states in $F \setminus G$ (none here) have a double plain line, states in $G \setminus F$ have a double dashed line, and state in $F \cap G$ use a mixed dashed/plain style.

Emptiness check using TA A first difference between the BA and TA approaches appears in the product computation. Indeed, a testing automaton remains in the same state when the Kripke structure executes a stuttering step.

The emptiness check also requires a dedicated algorithm because there are two ways to accept an execution: Büchi acceptance or livelock acceptance. In the algorithm sketched by Geldenhuys and Hansen [8], a first pass is used with an heuristic to detect both Büchi and livelock acceptance cycles. Unfortunately, in certain cases this first pass fails to report existent livelock acceptance cycles. This implies that when no counterexample is found by the first pass, a second one is required to double-check for possible livelock acceptance cycles. These two passes are annoying when the property is satisfied (no counterexample) since the entire state-space has to be explored twice.

Optimizations Looking at Fig. 3 inspires two optimizations. The first one is based on the fact that the construction of testing automata described in previous section will generate a lot of bisimilar states such as $(Gb, \bar{a}b)$ and (Gb, ab) . This is because the construction considers all the elements of K that are compatible with Gb . Had the LTL formula been over $AP = \{a, b, c\}$, e.g., $(a \vee c)UGb$, then we would have had four bisimilar states: $(Gb, \bar{a}b\bar{c})$, $(Gb, \bar{a}bc)$, $(Gb, ab\bar{c})$, and (Gb, abc) . These state are *necessarily* isomorphic, because they only differ in a and c , some propositions that the formula Gb does not *observe*.

A more efficient way to construct the testing automaton (and to construct the automaton from Fig. 3b directly) would be to consider only the subset of atomic propositions that are observed by the corresponding state of the Büchi automaton or its descendants (if the state is labeled by an LTL formula, the atomic propositions occurring in this formula give an over-approximation of that set).

A second optimization relies on the fact any state that no part of a SCC (also called *trivial* SCC) can be added to F without changing the language of the automaton. This is true for the three kinds of automata. For instance on Fig. 3 the state $(aUGb, \bar{a}b)$ can be added to F . Since this state is not part of any cycle, it cannot occur infinitely often and therefore cannot change the accepted language of the automaton.

This change allows further simplifications by bisimulation: the state $(aUGb, \bar{a}b)$ is now obviously equivalent to the (Gb, b) state. Fig. 4 shows the resulting automaton. Note that putting any trivial SCC x in F before performing bisimulation could hinder the reduction if x was isomorphic to some state not in F . However if x has only successors in F , as in our example, then it can be put safely in F : indeed, it can only be isomorphic to an F -state, or to another trivial SCC that will be added to F . This condition is similar to the one used by Löding before minimizing deterministic weak ω -automata [14].

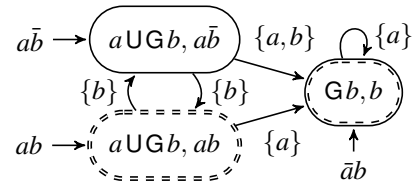


Fig. 4: Reduced TA for $aUGb$.

3 Experimentation

This section presents our experimentation of the various types of automata within our tool Spot [16]. We first present the Spot architecture and the way the variation on the model checking algorithm was introduced. Then we present our benchmarks (formulae and models) prior to the description of our experiments.

3.1 Implementation on top of Spot

Spot is a model-checking library offering several algorithms that can be combined to build a model checker [7]. Fig. 5 shows the building blocks we used to implement the three approaches. The TGBA and BA approaches share the same synchronized product and emptiness check, while a dedicated algorithms is required by the TA approach.

In order to evaluate our approach on “realistic” models, we decided to couple the Spot library with the CheckPN tool [7]. CheckPN implements Spot’s Kripke structure interface in order to build the state space of a Petri net on the fly. This Kripke structure is then synchronized with an ω -automaton (TGBA, BA, or TA) on the fly, and fed to the suitable emptiness check algorithm. The latter algorithm drives the on-the-fly construction: only the explored part of the product (and the associated states of the Kripke structure) will be constructed.

Constructing the state space on-the-fly is a double-edged optimization. Firstly, it saves memory, because the state-space is computed as it is explored and thus, does not need be stored. Secondly, it also saves time when a property is violated because the

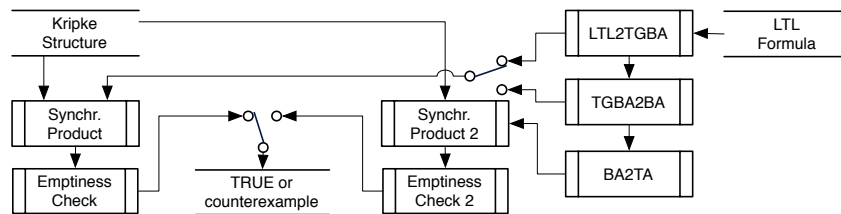


Fig. 5: The experiment’s architecture. Two command-line switches controls which one of the three approaches is used to verify an LTL formula on a Kripke structure.

emptiness check can stop as soon as it has found a counterexample. However, on-the-fly exploration is costlier than browsing an explicit graph: an emptiness check algorithm such as the one for TA [11] that does two traversals of the full state-space in the worst case (e.g. when the property holds) will pay twice the price of that construction.

In the CheckPN implementation of the Kripke structure, the Petri Net marking are compressed to save memory. The marking of a state has to be uncompressed every time we compute its successors, or when we compute the value of the atomic properties on this state. These two operations often occur together, so there is a one-entry cache that prevents the marking from being uncompressed twice in a row.

3.2 Benchmark Inputs

We selected some Petri net models and formulæ to compare these approaches.

Toy Examples A first class of four models were selected from the Petri net literature [1]: the flexible manufacturing system (FMS), the Kanban system, the dining philosophers, and the slotted-ring system. All these models have a parameter n . For the dining philosophers, and the slotted-ring, the model are composed of n identical 1-safe subnets. For FMS and Kanban, n only influences the number of tokens in the initial marking.

We chose values for n in order to get state space having between 2×10^5 to 3×10^6 nodes. The objective is to have comparable state spaces to be synchronized.

Case Studies The following two bigger models, were taken from actual cases studies. They come with some *dedicated* properties to check.

MAPK models a biochemical reaction: Mitogen-activated protein kinase cascade [12]. For a scaling value of 8 (that influences the number of tokens in the initial marking), it contains 22 places and 30 transitions. Its state space contains 6.11×10^6 states. The authors propose to check that from the initial state, it is necessary to pass through states *RafP*, *MEKP*, *MEKPP* and *ERKP* in order to reach *ERKPP*. In LTL:

$$\Phi_1 = \neg((\neg RafP) \cup MEKP) \wedge \neg((\neg MEKP) \cup MEKPP) \wedge \neg((\neg MEKPP) \cup ERKP) \wedge \neg((\neg ERKP) \cup ERKPP)$$

PolyORB models the core of the μ broker component of a middleware [13] in an implementation using a Leader/Followers policy [18]. It is a Symmetric Net and, since CheckPN processes P/T nets only, it was unfolded into a P/T net. The resulting net, for a configuration involving three sources of data, three simultaneous jobs and two threads (one leader, one follower) is composed of 189 places and 461 transitions. Its state space contains 61 662 states³. The authors propose to check that once a job is issued from a source, it must be processed by a thread (no starvation). It corresponds to:

$$\Phi_2 = G(MSrc_1 \rightarrow F(DOSrc_1)) \wedge G(MSrc_2 \rightarrow F(DOSrc_2)) \wedge G(MSrc_3 \rightarrow F(DOSrc_3))$$

Types of Formulæ As suggested by Geldenhuys and Hansen [8], the type of formula may affect the performances of the various algorithms. In addition to the formulæ Φ_1 and Φ_2 above, we consider two classes of formulæ:

³ This is a rather small value compared to MAPK but, due to the unfolding, each state is a 189-value vector. PolyORB with three sources of data, three simultaneous jobs and three threads would generate 1 137 096 states with 255-value vectors, making the experiment much too slow.

- *RND*: randomly generated LTL formulæ (without X operator). Since random formulæ are very often trivial to verify (the emptiness check needs to explore only a handful of states), for each model we selected only random formulæ that required to explore more than 2000 states with the TGBA approach.
- *WFair*: properties of the form $(\bigwedge_{i=1}^n GF p_i) \rightarrow \varphi$, where φ is a randomly generated LTL formula. This represents the verification of φ under the weak-fairness hypothesis $\bigwedge_{i=1}^n GF p_i$. The automaton representing such a formula has at least n acceptance conditions which means that the BA will in the worst case be $n + 1$ times bigger than the TGBA. For the formulæ we generated for our experiments we have $n \approx 3.19$ on the average.

All formulæ were translated into automata using Spot, which was shown experimentally to be very good at this job [19].

3.3 Results

Table 1 and 2 show how the three approaches deal with toy models and random formulæ (Table 1) and with toy models against WFair formulæ (Table 2). Table 3 shows the results of the two cases studies against random, weak-fairness, and dedicated formulæ.

These tables separate cases where formulæ are verified from cases where they are violated. In the former (left sides of the tables), no counterexample are found and the full state space had to be explored; in the latter (right sides) the on-the-fly exploration of the state space stopped as soon as the existence of a counterexample could be computed.

The numbers displayed in parentheses on both sides of the tables are the number of formulæ involved in the experiment. For instance (reading Table 2) we checked Kanban5 against 98 weak-fairness formulæ that had no counterexample, and against 102 weak-fairness formulæ that had a counterexample. The average and maximum are computed separately on these two sets of formulæ.

Column-wise, these tables show the average and maximum sizes (states and transitions) of: (1) the automata $A_{\neg\varphi_i}$ expressing the properties φ_i ; (2) the products $A_{\neg\varphi_i} \otimes A_M$ of the property with the model; and (3) the subset of this product that was actually explored by the emptiness check. For verified properties, the emptiness check of TGBA and BA always explores the full product so these sizes are equal, while the emptiness check of TA always performs two passes on the full product so it shows double values. On violated properties, the emptiness check aborts as soon as it finds a counterexample, so the explored size is usually significantly smaller than the full product.

The emptiness check values show a third column labeled “T”: this is the time (in hundredth of seconds, a.k.a. centiseconds) spent doing that emptiness check, including the on-the-fly computation of the subset of the product that is explored. The time spent constructing the property automata from the formulæ is not shown (it is negligible compared to that of the emptiness check). These tests were performed on a 64bit Linux system running on an Intel Core i7 CPU 960 at 3.20GHz, with 24GB of RAM. Running this entire benchmark with four tasks in parallel took us two days.

		Property verified (no counterexample)						Property violated (a counterexample exists)								
		Automaton		Full product		Emptiness check		Automaton		Full product		Emptiness check				
		st.	tr.	st.	tr.	st.	tr.	st.	tr.	st.	tr.	st.	tr.			
FMS5 (70)	TGBA	Avg	5.9	67.1	698 449	4 750 201	698 449	4 750 201	740	6.3	75.8	8 190 410	73 457 965	118 742	681 874	109
		Max	24	310	5 961 942	54 621 333	5 961 942	54 621 333	7 685	30	493	35 692 168	462 702 111	4 554 970	28 262 831	4 127
	BA	Avg	7.3	79.6	790 859	5 389 591	790 859	5 389 591	830	7.6	89.9	8 848 201	79 645 055	89 948	451 848	77
		Max	28	338	8 310 792	72 673 494	8 310 792	72 673 494	9 582	63	1 037	37 211 496	473 322 666	3 085 939	23 927 298	3 565
	TA	Avg	27.1	365.5	5 21 260	4 023 469	1 042 519	8 046 939	1 865	26.8	389.6	8 235 551	67 897 061	61 095	338 607	91
		Max	82	2 256	4 078 242	32 815 605	8 156 484	65 631 210	14 490	123	3 255	34 897 110	295 594 539	1 860 929	14 720 770	3 819
Kanban5 (100)	TGBA	Avg	5.2	48.5	852 364	7 279 249	852 364	7 279 249	909	7.0	71.6	7 126 650	77 809 374	47 984	237 295	33
		Max	27	264	6 694 184	70 465 136	6 694 184	70 465 136	8 373	22	292	21 715 730	241 387 835	1 604 560	11 177 672	1 510
	BA	Avg	6.1	56.3	852 493	7 279 889	852 493	7 279 889	910	8.6	87.6	8 041 841	87 518 994	36 085	194 392	25
		Max	29	296	6 694 184	70 465 136	6 694 184	70 465 136	8 335	38	472	23 997 065	270 130 066	1 628 283	11 232 778	1 513
	TA	Avg	20.2	227.2	6 51 299	6 074 858	1 302 598	12 149 717	2 451	29.7	368.3	7 162 575	70 438 470	17 766	141 630	29
		Max	114	1 858	6 409 984	62 033 608	12 819 968	124 067 216	25 344	134	2 221	17 551 016	175 769 251	1 163 547	10 736 232	2 217
Philo8 (100)	TGBA	Avg	6.1	87.0	219 303	1 232 080	219 303	1 232 080	257	7.3	99.2	637 670	4 950 129	36 161	168 189	37
		Max	20	338	830 533	6 366 282	830 533	6 366 282	1 172	27	360	1 489 852	16 311 100	63 4183	5 245 872	963
	BA	Avg	7.1	98.5	220 049	1 234 944	220 049	1 234 944	258	9.1	122.0	737 638	5 767 111	29 216	105 082	25
		Max	21	367	830 533	6 366 282	830 533	6 366 282	1 174	38	604	3 005 819	32 843 222	344 134	1 308 577	317
	TA	Avg	30.9	541.9	148 562	1 029 393	297 124	2 058 786	662	36.6	619.0	636 866	4 677 877	18 925	89 670	33
		Max	110	3 123	554 335	3 980 981	1 108 670	7 961 962	2 472	160	3 225	2 491 222	20 365 681	217 114	1 549 281	497
Ring6 (100)	TGBA	Avg	5.4	58	476 612	2 940 953	476 612	2 940 953	564	7.0	90.4	1 702 969	11 452 375	144 848	694 019	136
		Max	18	236	4 162 012	45 176 784	4 162 012	45 176 784	7 181	20	385	5 172 800	35 474 194	11 729 951	7 407 167	1 401
	BA	Avg	6.3	65.7	494 077	3 012 946	494 077	3 012 946	582	8.5	109.2	1 865 260	12 543 141	117 181	576 625	110
		Max	22	326	4 378 216	46 903 064	4 378 216	46 903 064	7 683	25	401	5 211 769	43 250 640	1 323 327	8 460 521	1 584
	TA	Avg	22.6	310.0	379 088	2 163 360	758 175	4 326 721	1 329	33.8	540.6	1 697 686	10 029 775	68 807	368 600	113
		Max	122	2 382	2 232 820	14 106 432	4 465 640	28 212 864	8 130	141	3 531	4 891 128	28 812 656	946 951	5 415 785	1 726

Table 1: Comparison of the three approaches on toy examples with random formulae, when counterexamples do not exist (left) or when they do (right).

		Property verified (no counterexample)						Property violated (a counterexample exists)								
		Automaton		Full product		Emptiness check		Automaton		Full product		Emptiness check				
		st.	tr.	st.	tr.	st.	tr.	T	st.	tr.	st.	tr.	T			
FMS5 (37)	TGBA	Avg	3.1	26	5197375	43078717	5197375	43078717	6191	5.4	49.5	9935828	89550059	627618	3517626	559
		Max	7	104	9866094	91499667	9866094	91499667	13282	12	212	21413973	319212813	5865891	51379790	7313
	Avg	7.3	58.4	7325010	53471546	7325010	53471546	7708	11.8	112.6	17297219	154876145	651799	3894388	593	
BA	Max	35	526	11338161	103816053	11338161	103816053	13394	49	578	64477308	784721607	17345804	148875504	21435	
	Avg	36.0	361.1	3967433	31419765	7934866	62839531	14231	60.6	656.7	15339186	126259786	216321	1526860	364	
	Max	215	3460	9002196	70152851	18004392	140305702	31515	205	2985	47074692	415672995	3732706	30145223	7165	
TGBA	Avg	2.7	14.9	2730709	23071387	2730709	23071387	2788	3.5	25	5484209	55893401	526015	3049738	410	
	Max	7	56	8092182	78624126	8092182	78624126	10214	10	140	13900320	166038726	2895449	24460029	3005	
	Avg	5.9	31	3382871	26705745	3382871	26705745	3183	7.1	53.2	8408110	82426568	531367	3035376	415	
BA	Max	20	150	12307085	113079575	12307085	113079575	11962	30	354	23144848	300434051	6104368	43693336	6384	
	Avg	21.0	123.6	1923597	17403907	3847194	34807815	6891	32.8	281.9	6365280	61028298	146619	1200463	240	
	Max	108	1364	6677524	63784672	13355048	127569344	26651	187	2554	18114712	190516984	1163652	10736394	2146	
TGBA	Avg	3.0	19.1	191233	1072039	191233	1072039	225	4.1	40.9	388356	2836796	11526	22540	8	
	Max	10	72	961946	8584333	961946	8584333	1581	11	110	1106279	10139160	148028	667632	153	
	Avg	7.2	47.7	226231	1219657	226231	1219657	254	9.5	107.3	925540	6664879	13374	32724	10	
BA	Max	24	213	961946	8584333	961946	8584333	1577	29	459	3369900	24286322	290681	1107465	265	
	Avg	32.3	245.9	141303	969063	282607	1938127	615	68.0	839.7	898752	6458513	11212	24675	13	
	Max	128	1746	665509	5048600	1331018	10097200	3026	205	3027	2280459	16828197	99824	619861	200	
TGBA	Avg	3.5	21.3	362296	2072837	362296	2072837	413	3.7	37.3	903909	5518052	27114	105130	23	
	Max	10	98	2116458	13877156	2116458	13877156	2531	12	109	2573186	16268868	831566	4479900	929	
	Avg	7.2	44.9	436729	2370915	436729	2370915	476	8.6	92.8	2112826	12623603	39004	168105	35	
BA	Max	22	240	2868218	17192038	2868218	17192038	3168	37	528	6641645	42624886	1123128	5300114	145	
	Avg	30.3	220.1	329599	1831831	659198	3663661	1121	61.6	732.3	2166241	12573562	27645	141549	44	
	Max	154	2020	1658112	9402736	3316224	18805472	5629	237	3456	5113422	30167566	793363	4498438	1408	
Ring6 (100)																
Philo8 (100)																
Kanban5 (98)																
FMS5 (37)																
Ring6 (100)																
Philo8 (100)																
Kanban5 (102)																
FMS5 (163)																

Table 2: Comparison of the three approaches on toy examples with weak-fairness formulae, when counterexamples do not exist (left) or when they do (right).

4 Discussion

Although the state space of cases studies can be very different from random state spaces [17], a first look at our results confirms two facts already observed by Geldenhuys and Hansen using random state spaces [8]: (1) although the TA constructed from properties are usually a lot larger than BA, the average size of the full product is smaller thanks to the more deterministic nature of the TA. (2) For violated properties, the TA approach explores less states and transitions on the average than the BA.

We complete this picture by showing run times, by separating verified properties from violated properties, and by also evaluating the TGBA approach.

On verified properties, the results are very straightforward to interpret: the BA are slightly worse than the TGBA because they have to be degeneralized. In fact, the average number of acceptance conditions needed in random formulæ (Table 1 and 3) is so close to 1 that the degeneralization barely changes the sizes of the automata. With weak-fairness formulæ (Table 2 and 3), the number of acceptance conditions is greater, so TGBA are favored over BA. Surprisingly, both TGBA and BA, although they are not tailored to *stuttering-insensitive* properties like TA, appear more effective to prove that a *stuttering-insensitive* property is verified. In the three tables, although the full product of the TA approach is smaller than the other approaches, it has to be explored twice (as explained in section 2.3): the emptiness-check consequently explores more states and transitions. This double exploration is not enough to explain the big runtime differences. Two other subtler implementation details contribute to the time difference:

- To synchronize a transition of a Kripke structure with a transition (or a state in case of stuttering) of a TA, we must compute the symmetric difference $l(s) \oplus l(d)$ between the labels of the source and destination states. The same synchronization in the TGBA and BA approaches requires to know only the source label.

Computing these labels is a costly operation in CheckPN because Petri net marking are compressed in memory to save space. Although we implemented some (limited) caching to alleviate the number of such label computation, profiling measures revealed the TA approach was 3 times slower than the TGBA and BA approaches, but that labels were computed 9 times more.

- A second implementation difference, this time in favor of the TA approach, is that transitions of testing automata are labeled by elements of K , while transitions of TGBA and BA are labeled by elements of 2^K . That means that once $l(s) \oplus l(d) \in K$ has been computed, we can use a hash table to immediately find matching transitions of the testing automaton. In the TGBA and BA implementations, we linearly scan the list of transitions of the property automaton until we find one compatible with $l(s)$. The BA and TGBA approaches could be improved by replacing each transition labeled by an element of 2^K by many transitions labeled by an elements of K , and then using a hash table, but we have not implemented it yet.

In an implementation where computing labels is cheap, the run time should be proportional to the number of transitions explored by the emptiness check, so it is important not to consider only the run time provided by our experiments.

On violated properties, it is harder to interpret these tables because the emptiness check will return as soon as it finds a counterexample. Changing the order in which

non-deterministic transitions of the property automaton are iterated is enough to change the number of states and transitions to be explored before a counterexample is found: in the best case the transition order will lead the emptiness check straight to an accepting cycle; in the worst case, the algorithm will explore the whole product until it finally finds an accepting cycle. Although the emptiness check algorithms for the three approaches share the same routines to explore the automaton, they are all applied to different kinds of property automata, and thus provide different transition orders.

This ordering luckiness explains why the BA approach sometimes outperforms the TGBA approach: one very bad case is enough to bias the average case. For instance this occurred on the Philo8 model with random formulæ: the worst TGBA case explored 4 times more transitions than the BA case, although the full product was twice smaller.

We believe that the TA, since they are more deterministic, are less sensible to this ordering. They also explore a smaller state space on the average. This smaller exploration is not always tied a good runtime because of the extra computation of labels discussed previously. Again, looking at the average number of transition explored by the emptiness check indicates that the TA approach would outperform the others if the computation of labels was cheap.

Finally in all of our experiments the TA approach has always found the counterexample in the first pass of the emptiness check algorithm. This supports Geldenhuys and Hansen’s claim that the second pass was seldom needed for debugging (less than 0.005% of the cases in their experiments [8]).

5 Conclusion

Geldenhuys and Hansen have evaluated the performance of the BA and TA approaches with small random Kripke structures checked against LTL formulæ taken from the literature [8]. In this work, we have completed their experiments by using actual models and different kinds of formulæ (random formulæ not trivially verifiable, random formulæ expressing weak-fairness formulæ, and a couple of real formulæ), by evaluating the TGBA approach, and by distinguishing violated formulæ and verified formulæ in the benchmark.

For verified formulæ, we found that the state space reduction achieved by the TA approach was not enough to compensate for the two-pass emptiness check this approach requires. It is therefore better to use the TGBA approach to prove that a *stuttering-insensitive* formula is verified and TA approach in an earlier “debugging phase”.

When the formulæ are violated, the TA approach usually processes less transitions than the BA approach and TGBA to find a counterexample. This approach should therefore be a valuable help to debug models (i.e. when counterexamples are *expected*). This is especially true on random formulæ. With weak-fairness formulæ, generalized automata are advantaged and are able to beat the TA on the average in 3 of our 6 examples (Philo8, Ring6, PolyORB 3/2/2).

Future work We plan to combine the ideas of TA and TGBA approaches. We believe it would be interesting to have testing automata with transition-based generalized acceptance conditions. We think the LTL translation algorithm we use to produce TGBAs could be adjusted to produce such automata directly.

References

1. G. Ciardo, G. Lüttgen, and R. Siminiceanu. Efficient symbolic state-space construction for asynchronous systems. In *Proc. of ICATPN'00*, vol. 1825 of *LNCS*, pp. 103–122. Springer.
2. J. Cichoń, A. Czubak, and A. Jasiński. Minimal Büchi automata for certain classes of LTL formulas. In *Proc. of DEPCOS'09*, pp. 17–24. IEEE Computer Society.
3. J.-M. Couvreur. On-the-fly verification of temporal logic. In *Proc. of FM'99*, vol. 1708 of *LNCS*, pp. 253–271. Springer.
4. J.-M. Couvreur. Un point de vue symbolique sur la logique temporelle linéaire. In *Actes du Colloque LaCIM 2000*, vol. 27 of *Publications du LaCIM*, pp. 131–140. Université du Québec à Montréal, Aug. 2000.
5. J.-M. Couvreur, A. Duret-Lutz, and D. Poitrenaud. On-the-fly emptiness checks for generalized Büchi automata. In *Proc. of SPIN'05*, vol. 3639 of *LNCS*, pp. 143–158. Springer.
6. J. Dallien and W. MacCaull. Automated recognition of stutter-invariant LTL formulas. *Atlantic Electronic Journal of Mathematics*, (1):56–74, 2006.
7. A. Duret-Lutz and D. Poitrenaud. SPOT: an extensible model checking library using transition-based generalized Büchi automata. In *Proc. of MASCOTS'04*, pp. 76–83. IEEE Computer Society Press.
8. J. Geldenhuys and H. Hansen. Larger automata and less work for LTL model checking. In *Proc. of SPIN'06*, vol. 3925 of *LNCS*, pp. 53–70. Springer.
9. J. Geldenhuys and A. Valmari. Tarjan's algorithm makes on-the-fly LTL verification more efficient. In *Proc. of TACAS'04*, vol. 2988 of *LNCS*, pp. 205–219. Springer.
10. D. Giannakopoulou and F. Lerda. From states to transitions: Improving translation of LTL formulae to Büchi automata. In *Proc. of FORTE'02*, vol. 2529 of *LNCS*, pp. 308–326.
11. H. Hansen, W. Penczek, and A. Valmari. Stuttering-insensitive automata for on-the-fly detection of livelock properties. In *Proc. of FMICS'02*, vol. 66(2) of *Electronic Notes in Theoretical Computer Science*. Elsevier.
12. M. Heiner, D. Gilbert, and R. Donaldson. Petri nets for systems and synthetic biology. In *Proc. of SFM'08*, vol. 5016 of *LNCS*, pp. 215–264. Springer.
13. J. Hugues, Y. Thierry-Mieg, F. Kordon, L. Pautet, S. Barrir, and T. Vergnaud. On the formal verification of middleware behavioral properties. In *Proc. of FMICS'04*, vol. 133 of *Electronic Notes in Theoretical Computer Science*, pp. 139–157. Elsevier.
14. C. Löding. Efficient minimization of deterministic weak ω -automata. *Information Processing Letters*, 79(3):105–109, 2001.
15. S. Miyano and T. Hayashi. Alternating finite automata on ω -words. *Theoretical Computer Science*, 32:321–330, 1984.
16. MoVe/LRDE. The Spot home page: <http://spot.lip6.fr>, 2011.
17. R. Pelánek. Properties of state spaces and their applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 10(5):443–454, 2008.
18. I. Pyrali, M. Spivak, R. Cytron, and D. C. Schmidt. Evaluating and optimizing thread pool strategies for RT-CORBA. In *Proc. of LCTES'00*, pp. 214–222. ACM.
19. K. Y. Rozier and M. Y. Vardi. LTL satisfiability checking. In *Proc. of SPIN'07*, vol. 4595 of *LNCS*, pp. 149–167. Springer.
20. S. Schwoon and J. Esparza. A note on on-the-fly verification algorithms. In *Proc. of TACAS'05*, vol. 3440 of *LNCS*. Springer.
21. R. Sebastiani and S. Tonetta. "more deterministic" vs. "smaller" Büchi automata for efficient LTL model checking. In *Proc. of CHARME'03*, vol. 2860 of *LNCS*, pp. 126–140. Springer.
22. H. Tauriainen. *Automata and Linear Temporal Logic: Translation with Transition-based Acceptance*. PhD thesis, Helsinki University of Technology, Espoo, Finland, Sept. 2006.
23. M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Proc. of Banff'94*, vol. 1043 of *LNCS*, pp. 238–266. Springer.

When Graffiti Brings Order^{*}

Alban Linard and Didier Buchs

Université de Genève, Centre Universitaire d'Informatique
7 route de Drize, 1227 Carouge, Suisse

Abstract. Most researchers use *Petri nets* as a formal notation with mathematically defined semantics. Their graphical part is usually only seen as a notation, that does not carry semantics. Contrary to this tradition, we show in this article that, when created by a human, there is inherent semantics in the positions of places, transitions and arcs. We propose to use the full definition of *Petri nets*: whereas they have been deteriorated to their mathematically defined part only, their graphical information should be considered in their definition.

1 Introduction

Graphical information of *Petri nets* usually does not appear in their formal definitions. For instance, Figure 1 presents the traditional graphical *Petri net* representation of two dining philosophers, a textual, and a mathematical representation of the same *Petri net*. In research, we almost always consider them as equivalent. The graphical representation is used by the modeler, or for figures in articles, the textual representation is an example of input format for a tool, and the mathematical representation is used for scientific publications.

Are all these representations really equivalent? When we ask researchers in *Petri nets*, they often believe it. But their equivalence is only an assumption, it might thus be false. Why do people strive to maintain layouts in the model transformation field, for instance in [1]? Our doubts for *Petri nets* are exposed in Dialogue 1, through a fictional dialogue between two elements of a *Petri net*:

MASTER TRANSITION: Why does graphical information of *Petri nets* disappear in their formal definitions?

PLACEHOPPER: Because graphical information has no semantics.

MASTER TRANSITION: If there is no semantics in graphical information, why are *Petri nets* represented graphically?

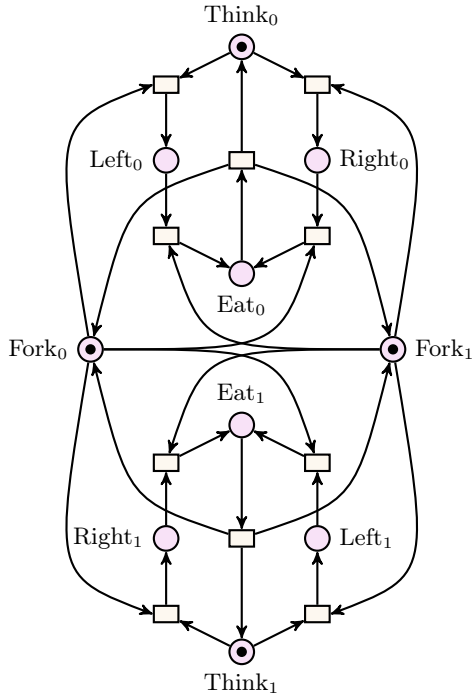
PLACEHOPPER: Because it helps the modelers to write and understand the models.

MASTER TRANSITION: But how does graphical information help the modelers if it has no semantics?

Dialogue 1: Question of PLACEHOPPER

Throughout this article, we propose to investigate the problem raised by MASTER TRANSITION, by extracting semantics from the graphical part of *Petri*

^{*}Thanks to Matteo Risoldi for proposing this title.



Places
 Think0, Think1 = 1
 Fork0, Fork1 = 1
 Left0, Right0, Eat0 = 0
 Left1, Right1, Eat1 = 0

Transitions
 Think0 + Fork0 → Left0
 Think0 + Fork1 → Right0
 Left0 + Fork1 → Eat0
 Right0 + Fork0 → Eat0
 Eat0 → Fork0 + Fork1 + Think0
 Think1 + Fork1 → Left1
 Think1 + Fork0 → Right1
 Left1 + Fork0 → Eat1
 Right1 + Fork1 → Eat1
 Eat1 → Fork0 + Fork1 + Think1

$$PN = \langle P, TA \rangle$$

$$P = \left\{ \begin{array}{l} Think_0, Left_0, Right_0, Eat_0, Fork_0 \\ Think_1, Left_1, Right_1, Eat_1, Fork_1 \end{array} \right\}$$

$$T = \left\{ \begin{array}{l} r_0, s_0, t_0, u_0, v_0 \\ r_1, s_1, t_1, u_1, v_1 \end{array} \right\}$$

$$Pre =$$

	Think ₀	Left ₀	Right ₀	Eat ₀	Fork ₀	Think ₁	Left ₁	Right ₁	Eat ₁	Fork ₁
r ₀	1	0	0	0	1	0	0	0	0	0
s ₀	1	0	0	0	0	0	0	0	0	1
t ₀	0	1	0	0	0	0	0	0	0	1
u ₀	0	0	1	0	1	0	0	0	0	0
v ₀	0	0	0	1	0	0	0	0	0	0
r ₁	0	0	0	0	0	1	0	0	0	1
s ₁	0	0	0	0	1	1	0	0	0	0
t ₁	0	0	0	0	1	0	1	0	0	0
u ₁	0	0	0	0	0	0	0	1	0	1
v ₁	0	0	0	0	0	0	0	0	1	0

$$Post =$$

	Think ₀	Left ₀	Right ₀	Eat ₀	Fork ₀	Think ₁	Left ₁	Right ₁	Eat ₁	Fork ₁
r ₀	0	1	0	0	0	0	0	0	0	0
s ₀	0	0	1	0	0	0	0	0	0	0
t ₀	0	0	0	1	0	0	0	0	0	0
u ₀	0	0	0	1	0	0	0	0	0	0
v ₀	1	0	0	0	1	0	0	0	0	1
r ₁	0	0	0	0	0	1	0	0	0	0
s ₁	0	0	0	0	0	0	0	1	0	0
t ₁	0	0	0	0	0	0	0	0	1	0
u ₁	0	0	0	0	0	0	0	0	1	0
v ₁	0	0	0	0	1	1	0	0	0	1

Fig. 1: Graphical, textual and mathematical representations of a Petri net

nets. This semantics is unclear, as it depends on the modeler and the model. Thus, we check that the extracted semantics can be useful to improve model checking performance of the *Petri nets*.

Section 2 presents the methodology of this work: how semantics extracted from the graphical information is used to improve model checking performance. Then we propose to use alignments of places and transitions in Section 3. As this only information is not always sufficient to improve model checking performance, we also propose to use graphical distances in Section 3. Then we propose in Section 4 to use surfaces delimited by arcs, for models with no alignments. The approach is generalized to colored *Petri nets* in Section 5. As we cannot give semantics to graphical information of all models, two counter-examples are provided in Section 6. Section 7 discusses about when and how graphical information can be used, before conclusion in Section 8.

2 How do we assess the quality of semantics extracted from graphical information?

The Software Modeling and Verification Group has developed the **Algebraic Petri Net Analyzer (ALPiNA)** [2], a model checker for Algebraic Petri nets. For efficient state space computation, this tool is based on **Decision Diagrams (DDs)** [3,4]. In this approach, a **Decision Diagram**, which is a particular kind of **Directed Acyclic Graph**, represents the state space. A node in the graph, called “variable”, represents the marking of each color for each place. When using DDs, the efficiency highly depends on the variable order in the graph [5].

By default, ALPiNA has rather good computation times [6], but its efficiency can be improved by orders of magnitude when the user provides “clustering” information, to group related variables. It is given in a textual notation. For instance, Listing 1 shows a clustering for the **Philosophers** model¹. Variables for the black token ● in places **Think0 .. Eat0** are put in the same cluster (group of variables) **c0**. The same applies for ● in **Think1 .. Eat1**, put in cluster **c1**.

```

Clusters c0, c1;
Rules
  cluster of ●
    in Think0, Left0, Right0, Fork0, Eat0
    is c0;
  cluster of ●
    in Think1, Left1, Right1, Fork1, Eat1
    is c1;
c0 < c1;

```

Listing 1: Example of clustering

Several articles already give heuristics to infer variable orders from *Petri nets* [7]. Clustering is less precise, as it only defines groups of variables, and

¹ This is not the exact syntax used in ALPiNA, but a very near one.

groups order. It does not define the order of variables within each group. However, AIPiNA also allows to define an order over places.

In this article, we propose to use the graphical information of Petri nets to define the clustering, together with ordering of the variables in the clusters. We try to group together tokens and places that belong to the same process.

To assess the inferred clustering, we check if it improves the model checker performances. Instead of using AIPiNA, we use PNxDD [8]. It is another Decision Diagram-based model checker that provides several clustering and ordering algorithms, contrary to AIPiNA. By using it, we can easily compare our approach with these algorithms. Because they are simpler, we begin our experiments with Place/Transition Petri nets and then use a Symmetric Petri net.

All the examples in this article are taken from the Model Checking Contest of the SUMo 2011 workshop. Choosing models that were not created by our group allows us to avoid a possible bias that may be introduced if we had too much knowledge about the models.

For each model, we compare the efficiency of our graphical clustering and ordering with fully random ones (500 random clusterings and orderings for each clustering proposed in this article). This benchmark shows the experimental probability of obtaining clustering and ordering with the same efficiency. Then we compare our graphical clustering and ordering with all the algorithms implemented in PNxDD. It shows how we perform against several existing algorithms.

The results are not intended to show that our approach is more efficient than other ordering algorithms, for instance those used by [9]. We provide them primarily to assess how much semantics is put in the graphical information of Petri nets, and also to assess if the semantics we extract makes sense. Therefore, we propose throughout this article informal ways to extract semantics from the graphical representation of Petri nets, instead of well defined algorithms.

We provide a VirtualBox image of the tools and scripts used in the benchmarks at <http://smv.unige.ch/members/dr-alban-linard/sumo2011-when-graffiti-brings-order-image> (use login/pass: sumo2011 for the Linux session).

3 Using alignment of places and transitions

We start the exploration of usual graphical semantics in Petri nets with the Flexible Manufacturing System (FMS) model. It is a Place/Transition Petri net represented in Figure 2. Initial marking is not shown as it is irrelevant in our approach. Note that this model is parameterized by its initial marking. The full model is presented in the SUMo 2011 workshop.

3.1 Alignment only

Even with absolutely no knowledge of the modeled system, it is obvious that the PN has vertically aligned places and transitions, linked by arcs in the alignment. Alignment, either vertical or horizontal, is one usual way for the modelers to show graphically the processes. Figure 2a shows the four vertical

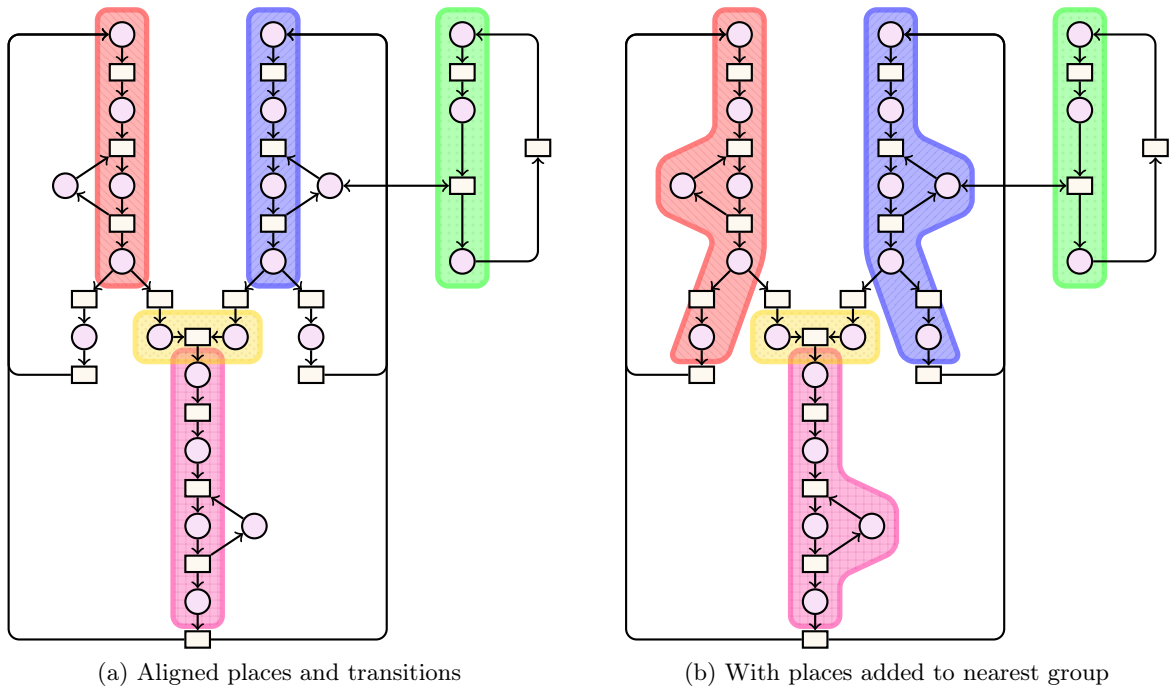


Fig. 2: Flexible Manufacturing System

alignments of places and transitions, and one horizontal alignment. Thus, maximal non-overlapping chains of consecutive and aligned places and transitions define groups. All the places of each group can be put in a same cluster. From this example, we can state Assumption 1:








Assumption 1 *Aligned places and transitions may correspond to a process.*

3.2 Alignment with graphical distance

After applying Assumption 1, some places do not belong to any group. Usually, the modelers use them either to represent shared data or for local data. In Figure 2a, the horizontal alignment shows a shared data. The remaining marked places are each near the middle of a cluster, and thus are likely to represent local data. On the contrary, the remaining unmarked places are found at the extremity of the clusters, and thus they can either represent the processes or local data. So, after defining the groups, we extend them using Assumption 2:

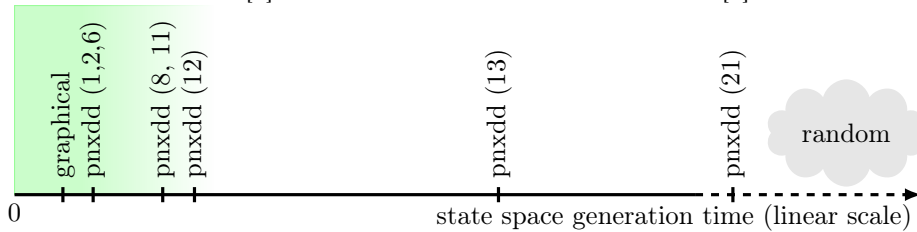
Assumption 2 *Local data of a process is more likely to be near its process.*

Figure 2b shows the result of application of Assumption 2 to the groups of Figure 2a. Each place is added to the nearest group it is related to, where nearest means graphical distance. For instance, the unmarked places are added to the vertical groups instead of the horizontal one, because they are linked to transitions aligned with the vertical clusters.

We also define ordering of the places inside the clusters and ordering of the clusters. Inside each cluster, the places are ordered from top to bottom or from left to right. The local data places are put randomly before or after the nearest process place. To order the clusters, we also chose graphical distance. In the `FMS` model,  and  are next to each other.  is far from all other clusters, but its nearest neighbor is . As  is between  and , we get the following order for the clusters:



The results of the benchmarks for the `FMS 50`, *i.e.*, with initial marking parameter set to 50. All random clusters and orders were too slow, and thus could not finish computation. Graphical clustering and ordering gives better results than all algorithms implemented in `PNXDD`. We give only the number of the algorithm, instead of its name, for a concise diagram. The corresponding algorithm names are found in [9] and in the documentation of the tool [8].



4 Using surfaces

Unlike the `fms` model, the `kanban` model shown in Figure 3 contains almost no alignments. The modeler has used different graphical semantics for this model.

For the human eye, the `kanban` model is composed of four components, each one with four places. All components have rather similar shapes. They differ by some arcs and also by a symmetry. Detecting these small differences is not a trivial task.

The components are visible because the arcs draw the shapes of the components. For instance, the modeler used arcs with two inflexion points where he could have drawn a direct arc. From this example, we can deduce Assumption 3:

Assumption 3 *Arcs may draw shapes, the surface of which may contain related places.*

We investigate different ways to use the shapes. First, Figure 3a identifies three different components in the model. Each one is found by searching maximal surfaces in the graphical `Petri net`. When two surfaces are linked by a place, they are merged, as one place can only belong to one cluster.

The maximal surfaces approach can lead to huge clusters. The worst case is when the whole `Petri net` is enclosed by arcs, and thus all its places are in the same cluster. We thus propose a second way to search surfaces. Instead of identifying maximal surfaces, we can use minimal surfaces. They are defined

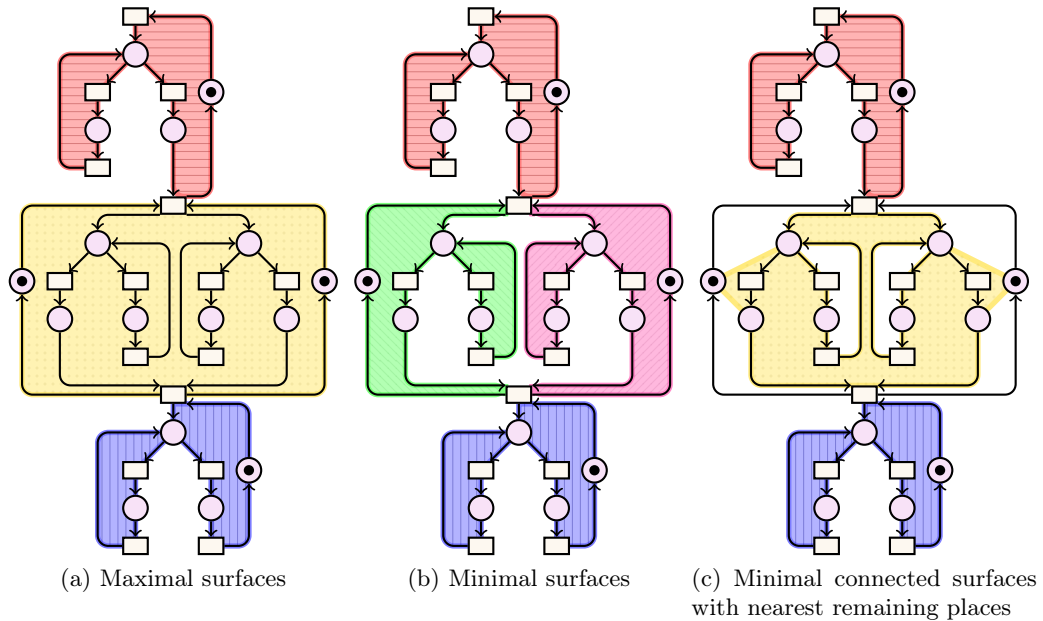
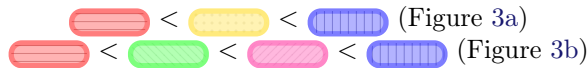


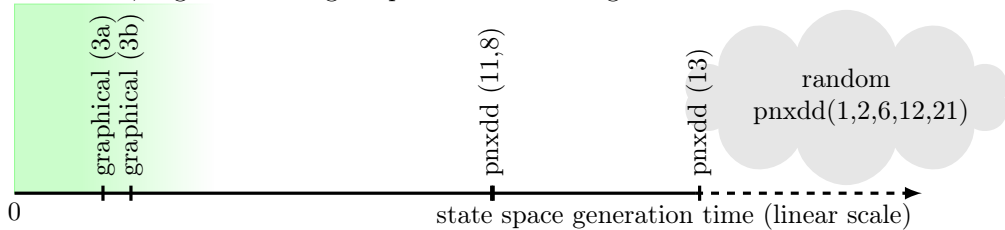
Fig. 3: Kanban System

as the surfaces that do not enclose another surface. Figures 3b and 3c show the Kanban model with minimal surfaces. In Figure 3b, the choice of minimal surfaces to consider leads to four clusters, whereas in Figure 3c another choice leads to three clusters. As the surfaces that touch the same place must be merged, we have to choose either Figure 3b or Figure 3c. Otherwise, the three surfaces in the middle block are merged, and the clustering is equivalent to the one of Figure 3a.

We compare Figure 3a and Figure 3b in the benchmarks. Places are ordered inside the clusters by following the boundaries of minimal surfaces. Clusters are ordered by graphical proximity, giving the clusters below:



The results of the benchmarks for the Kanban model are given below. All random clusters and orders were again too slow, and thus could not finish computation. The same applies to some algorithms of PNxDD (1,2,6,12,21). Whereas graphical clustering shows a small improvement over existing algorithms for the FMS model, it gives here huge improvements over algorithms in PNxDD.



5 Extension to colored Petri nets

The **Shared Memory** model of the **SUMo 2011 workshop** is given in Figure 4. It is a **Symmetric Petri net**, which is a particular kind of colored **Petri net**. Of course, our approach can be extended to the unfolded **Place/Transition Petri net** equivalent to a colored **Petri net**, when this equivalence exists. But the unfolding must then generate positions (possibly in three or more dimensions) to avoid superposition of unfolded places and transitions.

Without unfolding, we can still in some cases use graphical information to define clusters. Clusters for colored **Petri nets** are groups of unfolded places, *i.e.*, places together with colors. Two policies can coexist:

- Grouping in the same cluster all colors for one place,
- Distributing in its own cluster each color for one place.







The first policy works usually well for places representing shared data, whereas the second one leads usually to good results for places representing processes or data local to a process. Assumptions 4 and 5 describe these two policies.

Assumption 4 *One place related to other places using different variables may represent a shared data.*





Assumption 5 *Several places related to each other using the same variable, or equal ones, may represent a process.*

In the **Shared Memory** model, both Assumption 1 and Assumption 3 can be applied, as we find both aligned places and transitions and surfaces.

5.1 Using surfaces


To consider surfaces, we use Assumption 3. Its clustering is given in Figure 4a. We divided the model in four minimal surfaces. As there are places common to several surfaces, we obtain two groups. All places (a, b, c, e) in the  group have the same domain, except one (e) that is not colored (shown using ). We create one cluster for each color , , , or , as shown in Listing 2. Place e can only belong to one cluster, as its domain is the black token \bullet . Thus, we also create its own separate cluster.

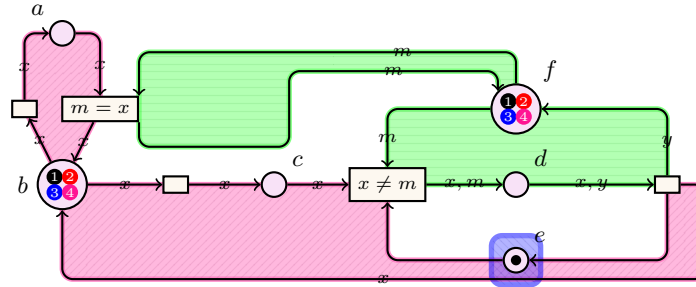
```

Clusters c, c1, c2, c3, c4;
Rules
  cluster of  $\bullet$  in  $e$  is c;
  cluster of  in  $a, b, c$  is c1;
  cluster of  in  $a, b, c$  is c2;
  cluster of  in  $a, b, c$  is c3;
  cluster of  in  $a, b, c$  is c4;

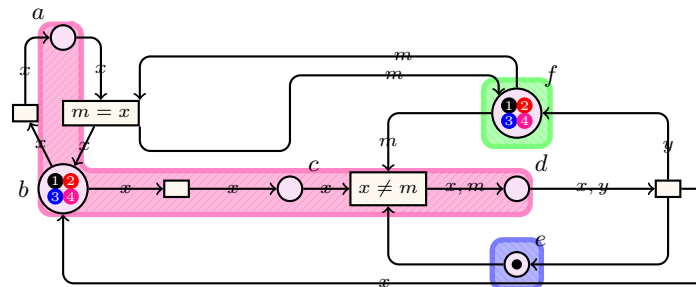
```

Listing 2: Clustering using surfaces for places a, b, c, e

The  group contains two places: d and f . Place f has domain $\{\bullet, \text{red}, \text{blue}, \text{pink}\}$, whereas the domain of the place d is a couple of colors. For each color, we create one cluster. Listing 3 completes Listing 2 with the new clusters.



(a) Minimal surfaces and Identity, with nearest Identity



(b) Alignment and Identity, with nearest Identity

Fig. 4: Shared Memory

Each token in place f is put in the cluster of its color. In place d , each token is a couple of colors. We have to choose which component of the couple predominates. Here, there is no obvious choice, so we choose the first element. Thus all tokens in place d are put in the cluster of the color of the first component.

```

Clusters d1, d2, d3, d4;
Rules
cluster of ● in f is d1;
cluster of ● in f is d2;
cluster of ● in f is d3;
cluster of ● in f is d4;
cluster of {●} × {●, ●, ●, ●} in d is d1;
cluster of {●} × {●, ●, ●, ●} in d is d2;
cluster of {●} × {●, ●, ●, ●} in d is d3;
cluster of {●} × {●, ●, ●, ●} in d is d4;
c < c1 < d1 < c2 < d2 < c3 < d3 < c4 < d4;
    
```

Listing 3: Clustering using surfaces for places d, f

5.2 Using alignments

Using Assumption 1 gives a totally different result. It is shown in Figure 4b. Listing 4 shows the clustering obtained using Assumption 1.

We first identify aligned places and transitions: places b, c, d belong to one group. Places b and c have the same colored domain, whereas the tokens of place d are couples of colors.

We then try to add to the group what seems local data or part of the processes. All places a, f, e are near the group. Place a is linked to the group using only variable x . This clearly shows that an identity is passed along the arcs. So, this place is added to the group.

Place e contains black tokens. Because of this domain, it cannot be added in the clusters of colors, so we create its own cluster, as in Listing 2.

The last place is f . The variables that relate this place to d are used in the second part of the couple, whereas the variables that relate d to the places in its group are in the first part. So, f probably represents a shared data. We thus put place f in its own cluster, for all colors.

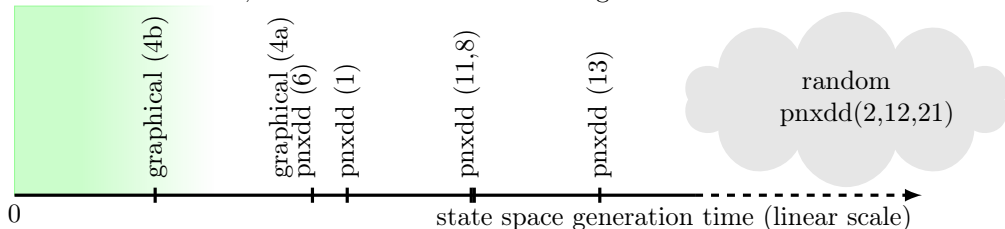
```

Clusters c, c1, c2, c3, c4, f;
Rules
cluster of ● in e is c;
cluster of ● in a,b,c is c1;
cluster of ● in a,b,c is c2;
cluster of ● in a,b,c is c3;
cluster of ● in a,b,c is c4;
cluster of ● in f is f;
cluster of ● in f is f;
cluster of ● in f is f;
cluster of ● in f is f;
cluster of {●} × {●,●,●,●} in d is c1;
cluster of {●} × {●,●,●,●} in d is c2;
cluster of {●} × {●,●,●,●} in d is c3;
cluster of {●} × {●,●,●,●} in d is c4;
c < c1 < c2 < c3 < c4 < f;
    
```

Listing 4: Clustering using alignment

Figure 4b shows the result of this analysis. Note that for each color ●, ●, ●, or ●, we create a cluster that contains all the highlighted places for this color only. For place d , each cluster contains all possible colors for the second part of the Cartesian product.

We compare both clusterings below. For this third model, the state space generation could not finish using some algorithms of PNxDD (2,12,21). Graphical clustering and ordering is still better than all algorithms implemented in PNxDD. As for other models, none of the random clusterings could finish.



6 When we fail at understanding the model

The **Model Checking Contest** of SUMo 2011 workshop uses seven models, three of which are **Place/Transition Petri nets** and the four others **Symmetric Petri nets**. In this article, we provide clustering for **Flexible Manufacturing System**, **Kanban** and **Shared Memory**. Two models are not discussed: **Philosophers** and **Token Ring**. The **Philosophers** model has been studied a lot for use with **Decision Diagrams**. Before writing this article, we already knew that the best order is when each philosopher and its left (or right) fork is put in its own cluster. Because of this knowledge, trying to find graphical information would be biased. The **Token Ring** model does not contain enough places (1) and transitions (2).

The two remaining models are **MAPK** (Figure 5) and **Peterson’s algorithm** (Figure 6). For them, we did not find how to give semantics to the graphical information. Note that these two models are considered as hard to understand by humans. This might be an explanation: because their modelers could not find a good disposition of places and transitions, we cannot give semantics to graphical information, and thus the models are hard to understand.

7 Discussion on the approach

We propose in this article to extract semantics from the graphical part of a **Petri net**. It works on some models, but does not work on some others. Moreover, all people do not always agree on how to identify the processes in the **Petri net**. This approach is thus fragile. We provide in this section some remarks on its applicability, and on other techniques that could be used along graphical information to improve clustering.

7.1 This approach depends on the school of modeling.

Depending on where we have studied, and where we work, we may not create models in the same way. The most obvious difference is putting aligned processes horizontally or vertically. But some people can also prefer to show processes using surfaces instead of alignments. We should investigate which representations are used, and where they are modeled.

A side effect is that every **Petri net** should define an author and the institutes where the author has studied and workse Using this information, giving semantics to graphical information could be enhanced. The “author” field is already present in the **Petri Net Markup Language** [10], but there is currently no traceability of the author’s institutions, nor of the school of modeling for this **Petri net**.

7.2 Graphical information cannot be processed easily

The **Petri Net Markup Language** handles absolute positions for the **Petri net** elements (places, transitions, labels. . .). The problem with absolute positioning is that analysis of the graphical information is not easy. Some other positioning schemes exist. For instance, **TikZ** [11] provides a way to define a position relative to another one (above, below, . . .). Such positions can be processed more easily.

7.3 Labels can also be used to detect components.

We use only graphical information on *Place/Transition Petri nets* in this article. For *Symmetric Petri nets*, we also use domains and variables on the arcs. Another interesting information to compute clustering is the labels of places and transitions. In *Place/Transition Petri nets*, they are often composed of a textual prefix, a textual suffix, and some numbers in the middle. These numbers correspond to colors in an equivalent colored *Petri nets*. Extracting the naming patterns of places can greatly enhance the clustering.

7.4 This approach may be redundant with graph analysis.

Ordering the *Decision Diagrams* used during the model checking of a *Petri net* has been explored for a long time. We can cite again [7], which is a survey of existing algorithms.

Instead of using the graphical information, they are usually based on the graph structure of the *Petri net*. These techniques, from the structural analysis field, can lead to very good results. Note that combining them with graphical information and naming analysis is often possible.

First, traps and siphons [12] are a way to detect components, and thus clusters. They can be computed efficiently, for instance in [13]. Invariants are another way to define clusters, but they show both processes and data. A good point is that they can even be computed in colored *Petri net* [14].

8 Conclusion

Throughout this article, we show *Petri nets* from the *Model Checking Contest of SUMo 2011 workshop*. For each one, we try to understand how their modelers put semantics in their graphical information. Whereas we cannot identify such information in some models (*MAPK* and *Peterson's algorithm*), it provides really good results for the others.

We extract semantics from several kinds of graphical information: the alignment of places and transition, the graphical proximity of places and groups, and the surfaces delimited by arcs. This semantics is used to define clustering and ordering of the places and tokens.

For all these examples, we use the graphical information to improve *symbolic model checking* of the *Petri net*. To do so, we extract clustering and ordering from the graphical part of each model. Using them, we compare the time taken to generate the state space with several clustering and ordering algorithms implemented in *PNXDD*, and with fully random clusters and orders. In all models where graphical clustering could be defined, we obtain improvements over *PNXDD*. No random clusters and orders could run in the time and memory limits, which shows that the semantics extracted from graphical information makes sense.

At the beginning of this work, we were only hoping to obtain improvements over some of the algorithms in *PNXDD*. Prior the benchmarks, we did no selection of the models, and of the inferred graphical semantics. Thus, the good results seem promising.

From these examples, two conclusions arise: (1) in education, *Petri nets* are always introduced as a graphical formalism. In research, they have been deteriorated into a mathematical only formalism, by losing graphical information; (2) graphical information in *Petri nets* has semantics, but unclear one.

We believe that, contrary to their usual presentation in scientific publications, *Petri nets* are truly a graphical formalism, and their mathematical representation is only a projection.

Further work should identify more graphical semantics, and the cases in which they apply. We may not be able to create algorithms to extract always semantics from graphical information. But when it is possible, a study with fully automatic algorithms should check that the inferred semantics can be used with no user knowledge of the models.

References

1. Sun, Y., Gray, J., Langer, P., Wimmer, M., White, J.: A WYSIWYG Approach for Configuring Model Layout using Model Transformations. In: DSM'10: Workshop on Domain-Specific Modeling @ Splash. (2010)
2. SMV Group: Algebraic Petri Nets Analyzer (2010) <http://alpina.unige.ch>.
3. Bryant, R.E.: Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers* **35**(8) (1986) 677–691
4. Couvreur, J.M., Encrenaz, E., Paviot-Adet, E., Poitrenaud, D., Wacrenier, P.A.: Data Decision Diagrams for Petri Net analysis. In: ICATPN '02: 23rd International Conference on Applications and Theory of Petri Nets. (2002) 101–120
5. Bollig, B., Wegener, I.: Improving the variable ordering of obdds is np-complete. *IEEE Transactions on Computers* **45**(9) (1996) 993–1002
6. Buchs, D., Hostettler, S., Marechal, A., Risoldi, M.: AlPiNA: A symbolic model checker. In: Petri Nets '10: International Conference on Theory and Applications of Petri nets. (2010) 287–296
7. Rice, M., Kulhari, S.: A survey of static variable ordering heuristics for efficient BDD/MDD construction. Technical report, UC riverside (2008)
8. Hong, S., Paviot-Adet, E., Kordon, F.: PNXDD Model Checkers – <https://srcdev.lip6.fr/trac/research/neoppod/> <https://srcdev.lip6.fr/trac/research/NEOPPOD/>.
9. Hong, S., Kordon, F., Paviot-Adet, E., Evangelista, S.: Computing a Hierarchical Static Order for Decision Diagram-Based Representation from P/T Nets. *ToPNoC: Transactions on Petri Nets and Other Models of Concurrency* (submitted) (2010)
10. Hillah, L.M., Kindler, E., Kordon, F., Petrucci, L., Trèves, N.: A primer on the Petri Net Markup Language and ISO/IEC 15909-2. *Petri Net Newsletter* **76** (2009)
11. Tantau, T., Feuersaenger, C.: TikZ ist kein Zeichenprogramm <http://www.ctan.org/tex-archive/graphics/pgf/>.
12. Murata, T.: Petri nets: Properties, analysis and applications. *Proceedings of the IEEE* **77**(4) (1989) 541–580
13. Barkaoui, K., Lemaire, B.: An Effective Characterization of Minimal Deadlocks and Traps in Petri nets Based on Graph Theory. In: 10th Int. Conf. on Application and Theory of Petri Nets ICATPN'89. (January 1989) 1–21
14. Couvreur, J.M., Haddad, S., Peyre, J.F.: Computation of generative families of positive semi-flows in two types of coloured nets. *International Conference on Theory and Applications of Petri Nets* (1991)

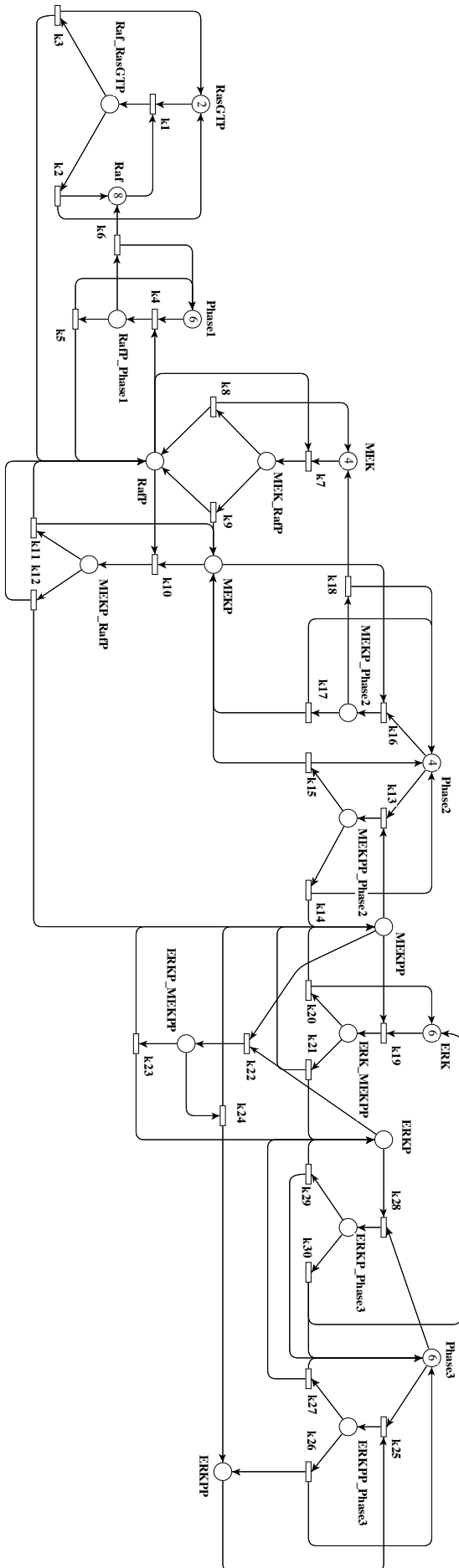


Fig. 5: MAPK: Mitogen-activated protein kinase kaskade

Author Index

B		
	Bashkin, Vladimir A.	33
	Bérard, Béatrice	17
	Ben Salem, Ala Eddine	65
	Boukala, Mohand Cherif	1
	Buchs, Didier	81
D		
	Duret-Lutz, Alexandre	65
F		
	Fronc, Lukasz	49
K		
	Kordon, Fabrice	65
L		
	Lime, Didier	17
	Linard, Alban	81
P		
	Petrucci, Laure	1
	Pommereau, Franck	49
R		
	Roux, Olivier H.	17
T		
	Thierry-Mieg, Yann	17

