Compositional Analysis of Discrete Time Petri nets

Yann Thierry-Mieg[1], Béatrice Bérard[1], Fabrice Kordon[1], Didier Lime[2], and Olivier H. Roux[2]

[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.

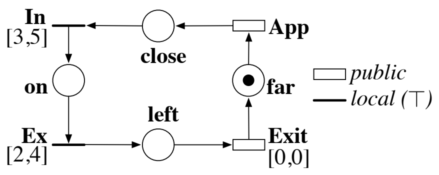
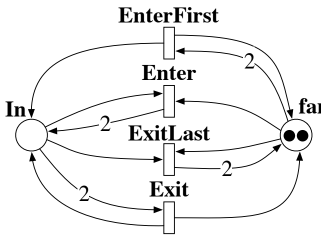


Fig. 1. Train Component (default time interval is $[0,\infty[$)



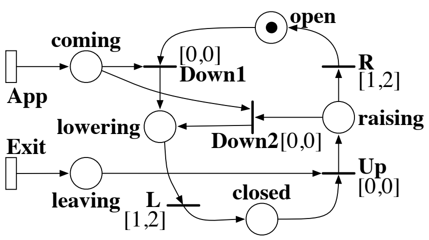


Fig. 2. Controller module for 2 trains **Fig. 3.** Gate module (default time interval is $[0,\infty[$)

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 $\mathcal{S}_{\mathcal{N}} = \langle S, s_0, A \cup \{\mathbb{1}\}, \rightarrow\rangle$ 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.*
- *$s_0 = \langle m_0, \overline{0}\rangle$, where the tuple $\overline{0}$ 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
  $$\begin{cases} \ell(t) = a \wedge enabled(m,t) \wedge v(t) \geq \alpha(t) \\ \text{and } m' \in fire(m,t) \\ \text{and } \forall t' \in Tr, v'(t') = \begin{cases} v(t') \text{ if } enabled(m',t') \wedge t \neq t' \\ 0 \text{ otherwise} \end{cases} \end{cases}$$

*The **delay** transition relation:*

$$\langle m,v\rangle \xrightarrow{\mathbb{1}} \langle m',v'\rangle \text{ iff } m'=m \text{ and for all } t\in Tr,$$
$$\begin{cases} enabled(m,t) \implies v(t)<\beta(t) & \textit{(urgent clocks prevent elapse)}\\ v'(t)=\begin{cases} v(t)+1 \text{ if } enabled(m',t)\wedge\beta(t)\neq\infty & \textit{(normal elapse)}\\ v(t)+1 \text{ if } enabled(m',t)\wedge\beta(t)=\infty\wedge v(t)<\alpha(t) & \\ & \textit{(only progress up to } \alpha\textit{)}\\ v(t) \text{ otherwise} & \end{cases} & \end{cases}$$

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 $A^\star$, with ε for the empty word and $\cdot$ (or no symbol) for the concatenation operation. We denote by $z.X, z.Y\ldots$ the element X (resp. $Y\ldots$) of a tuple $z=\langle X,Y,\cdots\rangle$.

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;*
- *$Succ: S\times A^\star\mapsto 2^S$ is a transition function satisfying: $\forall s\in S, Succ(s,\varepsilon)=\{s\}$.*

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

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

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

Definition 5 (Composite). *A **composite** is a tuple $C = \langle I, Sync, A, \lambda \rangle$ 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.*
- *$Sync \subset A_I$ is the finite set of synchronizations; A is a set of action labels, which contains the label $\top$ and $\lambda : Sync \mapsto A$ is the labeling function*

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

$s' \in Next_I(s, \sigma)$ *iff* $\forall i \in I, \pi_i(s') \in type(i).Succ(\pi_i(s), \pi_i(\sigma))$

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*

$$\begin{cases} \exists i \in I, \pi_i(s') \in type(i).Locals(\pi_i(s)) \wedge \forall j \in I, j \neq i, \pi_j(s') = \pi_j(s) \\ or\ \exists \sigma \in Sync, \lambda(\sigma) = \top, s' \in Next_I(s, \sigma) \end{cases}$$

- $Succ : S \times A^\star \mapsto 2^S$ *is defined for* $s, s' \in S,\ w = a_1 \cdots a_n \in A^\star$ *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: train	t_1: train	λ
App	ε	**App**
ε	**App**	**App**
Exit	ε	**Exit**
ε	**Exit**	**Exit**
$\mathbb{1}$	$\mathbb{1}$	$\mathbb{1}$

Fig. 4. Synchronization for 2 trains

tg: TrainGroup	*cc*: ContrGate	λ
Exit	**Exit**	$\top$
App	**App**	$\top$
$\mathbb{1}$	$\mathbb{1}$	$\mathbb{1}$

Fig. 5. Synchronization for the complete system

g: gate	*c*: controller	λ
App	**EnterFirst**	**App**
ε	**Enter**	**App**
Exit	**ExitLast**	**Exit**
ε	**Exit**	**Exit**
$\mathbb{1}$	$\mathbb{1}$	$\mathbb{1}$

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 $Succ(s,$ **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* $\mathcal{S} = \langle \tau, n, s_0, D, CSync \rangle$ *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 untimed 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 $A^\star$ for synchronisation: for a

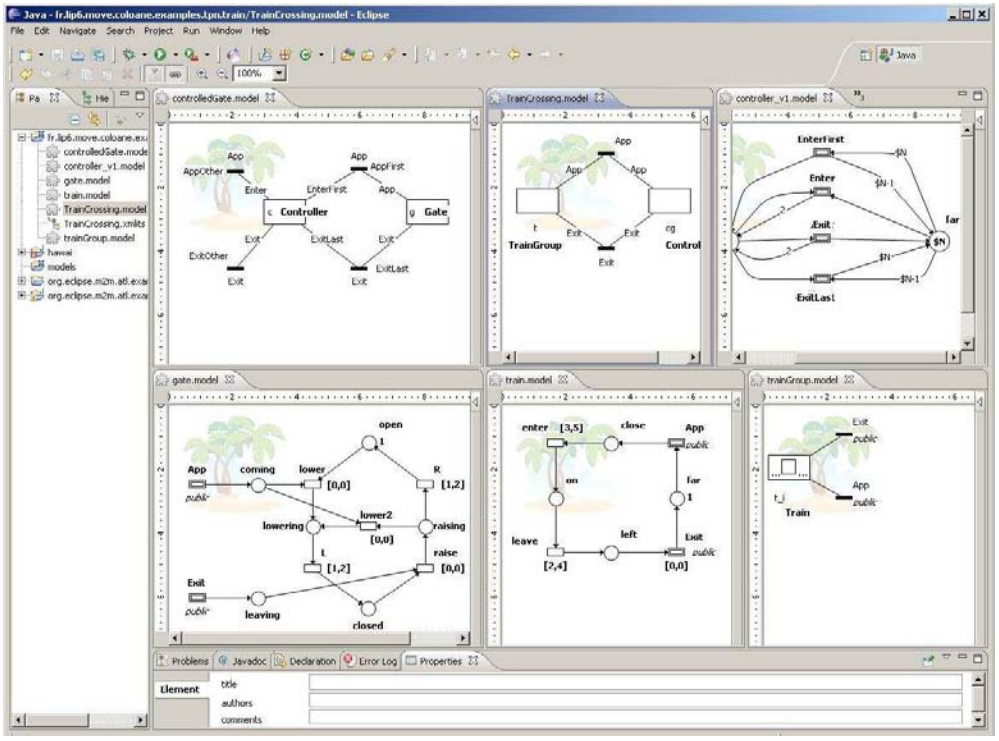


Fig. 9. Coloane based ITS modeler

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 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.

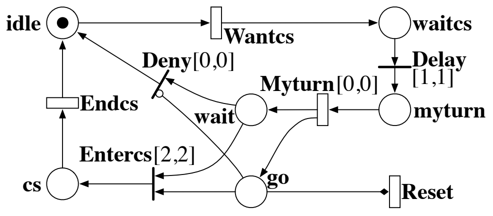


Fig. 10. Process type for Fischer.

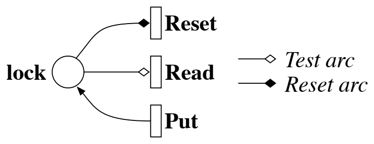


Fig. 11. The resource type.

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

Fig. 12. A Scalar set type ProcessGroup

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

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)

	Roméo			RED		UPPAAL/sym			Roméo/SDD		
N	*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	$1.17\ 10^{6}$
9	73 071	$1.77\ 10^{6}$	$3.72\ 10^{6}$	67	785 108	0.03	160	172	0.1	2 156	$6.20\ 10^{6}$
10	DNF	-	-	652	$2.35\ 10^{6}$	0.1	160	211	0.1	2 332	$3.26\ 10^{7}$
170	-	-	-	-	OOM	7 783	47 956	57 971	23	101 896	$2.27\ 10^{120}$
700	-	-	-	-	-	DNF	-	-	1391	$1.82\ 10^{6}$	$2.66\ 10^{491}$
730	-	-	-	-	-	-	-	-	1803	$2.33\ 10^{6}$	$2.58\ 10^{512}$

Train (*N* is the number of trains)

	Roméo			RED		UPPAAL/sym			Roméo/SDD		
N	*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	$4.83\ 10^{6}$
7	6 115	377 452	131 517	66	723 428	0.23	3 200	957	2.5	10 304	$6.28\ 10^{7}$
8	DNF	-	-	-	OOM	1	3 336	2 078	4	14 188	$8.16\ 10^{8}$
13	-	-	-	-	-	2 634	13 188	79 598	26	56 660	$3.02\ 10^{14}$
15	-	-	-	-	-	60 860	61 256		42	86 360	$5.11\ 10^{16}$
16	-	-	-	-	-	DNF	-	-	52	104 848	$6.65\ 10^{17}$
44	-	-	-	-	-	-	-	-	1143	$2.13\ 10^{6}$	$1.03\ 10^{49}$

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 Cynernetics*, 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.