Splitting Trees and Partition Refinement in Real-Time Model Checking

R.F. Lutje Spelberg\(^1\) and W.J. Toetenel\(^2\)

\(^1\) Thales Naval Systems, Hengelo, The Netherlands, rf.lutjespelberg@signaal.nl
\(^2\) Delft University of Technology, Faculty of Information Technology and Systems, Delft, The Netherlands, w.j.toetenel@its.tudelft.nl

Abstract. This paper discusses model checking of real-time systems. A novel aspect of our approach is the unconventional approach to deal with representing symbolic state spaces. The representation does not use a canonical form for representing symbolic nodes. Instead it applies an alternative representation based on splitting trees. Unlike often used canonical representations, this representation is tailored for the type of exploration algorithm that we apply, namely an algorithm based on partition refinement.

1 Introduction and overview

The main problem to be solved when building a model checker is dealing with the size of state spaces that are explored. To do so, often a symbolic model checking approach is adopted, especially if real-time systems are involved. The key characteristic of a symbolic model checker is that it explores the state space of systems in terms of symbolically represented sets of states, rather than single states. Constructing a model checker involves

- the selection of a syntactic and semantic basis for the operational modeling notation,
- selecting an appropriate declarative specification notation and
- finding a solution for the model checking exploration algorithms for reachability analysis and specification checking.

Modelling languages use often a graphical representation based on a finite automaton approach. The semantic basis for this class of modelling languages is in general a labelled transition system. The specification languages are in general variants of temporal logic. E.g. the popular model checker SPIN [14] applies a propositional linear temporal logic PLTL for requirements specification and a simple finite state automata that accepts infinite sequences of states, called Büchi automata for system modeling. SPIN is applied in the context of reactive systems with infinite behaviour. SMV [18] applies CTL [1], Computation Tree Logic, the branching time temporal logic counter part of PLTL. The modeling language is based on Kripke structures, a simple finite state automaton, where the successor of a state is now a possible set of states (in stead of one state in the case of Büchi automata). The model checker Uppaal [5] addresses real-time systems by introducing clocks (continuous variables) in the modelling notation called timed automata. The corresponding specification logic is a timed version of CTL, called TCTL. The techniques applied in our model checker PMC (parametric model checker) are explained in some detail in the remainder of this paper. The exploration algorithms perform simple checks on the state space that is constructed from both the model and specification. SPIN checks whether the product automaton of the system model and the negation of the specification is empty. SMV checks that the initial state \(s_0\) satisfies the given specification \(\Phi\). Uppaal checks that the partition class containing the initial state \(s_0\) satisfies the specification \(\Phi\) under the relation induced by the partition \(R\). Table 1 summarizes the approaches taken in four different model checking approaches.

<table>
<thead>
<tr>
<th>Model Checking Explorations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operational Modeling Language</td>
</tr>
<tr>
<td>Büchi Automata</td>
</tr>
<tr>
<td>Kripke Structure</td>
</tr>
<tr>
<td>Timed Automata</td>
</tr>
<tr>
<td>Timed Automata</td>
</tr>
</tbody>
</table>

Table 1: Model Checking Approaches
refinement have been described, but seem not to have lead to tools that match the performance of the reachability analysis-based tools. However, we have shown elsewhere that partition refinement is also a powerful approach to real-time model checking [17]. We will show that the key to applying partition refinement is the usage of a specific approach to represent symbolic state spaces. Traditionally, in symbolic model checking, canonical representations have been used. In particular, in non-real-time model checking, Binary Decision Diagrams (BDD’s) have proven to be very successful [18], while for real-time systems, Difference Bound Matrices (DBM’s) [9] have often been used. The application of a partition refinement technique enables the use of an unconventional approach in representing state sets. This paper discusses a symbolic model checking approach truly founded on splitting. It is a combination of an exploration algorithm that based on iterative splitting and a symbolic state space representation based on splitting trees. Although splitting trees have been used in some verification approaches [22, 20], they were to our knowledge never applied as means to represent symbolic state spaces.

We have applied the approach for parametric model checking of fair TCTL specifications. A tool was built that implements parametric fair-TCTL model checking based on this approach. Figure 1 summarizes our approach. The paper is further constructed as follows. Section 2 briefly discusses our modeling notation XTG and specification language F1CTL. Section 3 discusses the reachability algorithm based in partition refinement applied in our model checker. It further summarizes the application of splitting trees. Section 4 presents practical results, and section 5 presents our conclusions.

2 A simple model for systems

2.1 Modelling systems

For modeling parallel, distributed and real-time systems, we have defined an extension of timed automata [2], called XTG (eXTended Timed Graphs) [17]. Essentially, timed automata are finite state machines augmented with clocks and data. Clocks are non-negative real-valued variables that uniformly increase. The increasing of clocks models the progress of time. Conditions on clocks and data can be used to constrain (by guards on edges) and enforce (by invariants on locations) the execution of transitions. Upon execution, a transition may update clocks and data. Our key feature of XTG is that it provides a general form of urgency. Edges can either be marked normal or urgent, the latter indicating that they have to be executed immediately upon enabling, without letting time pass. This general form of urgency allows easy modeling of transitions that trigger on data and time conditions. An XTG system is defined as a set of graphs that synchronize and communicate by means of value passing and shared data. Value passing is done in the spirit of the synchronous value passing model of CCS [19]. This means that an edge labeled with a synchronization l must synchronize with an edge labeled with l?e, where l is a synchronization label (or channel name), e is value expression, and v is a variable. The effect is a value passing over the channel l as a consequence of which the value of e is assigned to the variable v. Furthermore processes can communicate through shared data.

Finally, the XTG notation allows the definition of parametric systems. We have taken a general approach to defining parametric systems. A subset of the (non-clock) variables of the system are designated as parameters. These parameters can be used anywhere in the system (i.e. in guards, invariants, and updates). The only difference between parameters and normal variables is that parameters are not given initial values. Note that in principle parameters can be assigned new values, but that this is generally not very useful.

The underlying semantic model for XTG models is that of timed structures. In literature other names can be found for similar models, e.g. timed Kripke structures, timed transition systems and labeled timed transition systems.
The notation \( s & s' \) is used to indicate a transition. Transitions are either time transitions or discrete transitions and are therefore labeled with a nonnegative real number. Time transitions model the elapsing of time and are thought to elapse with the transition. Discrete transitions model state changes and have a special label \( p \).

The XTG language is defined that is sufficient for defining operational semantics. Here, an abstract model of the data language will become very relevant when considering model checking strategies for systems. There is a strong connection between the expressiveness of the data language and the limitations of model checking techniques. Thus the expressiveness of the data language will depend on the possibilities of the model checking tool. We will come back to this later.

**Definition 2.2 (data language)**

Given a set of variables \( V \), a subset of clock variables \( V_C \subseteq V \), and a subset of parameters \( V_P \subseteq V \) with \( V_P \cap V_C = \emptyset \), we assume the following domains:

- \( \text{Val} \): possible values
- \( \text{Expr}_V \): value expressions over the set of variables \( V \), where \( \text{Bezpr}_V \subseteq \text{Expr}_V \) denotes the subset of boolean expressions

and the following semantical evaluation function

\[
\text{V}_V[I ] : \text{Expr}_V \rightarrow ((V \rightarrow \text{Val}) \rightarrow \text{Val})
\]

which evaluates value expressions over \( V \). Furthermore, the following assumptions are made:

- \( \mathbb{R}^{\geq 0} \subseteq \text{Val} \) and \( \{ \text{true}, \text{false} \} \subseteq \text{Val} \)
- \( \forall b \in \text{Bezpr}_V . \forall \rho \in (V \rightarrow \text{Val}) . \text{V}_V[b](\rho) \in \{ \text{true}, \text{false} \} \)

**Definition 2.3 (valuations)**

If \( V \) is a set of variables, then \( \text{Env}_V \) denotes the domain of valuations over \( V \): \( \text{Env}_V = V \rightarrow \text{Val} \). We use \( \rho \) to range over valuations.

If \( \rho \in \text{Env}_V \), \( x \in \text{Val} \) and \( v \in V \) then \( \rho[v \mapsto x] \) denotes an update of a variable, formally defined as follows:

\[
\rho[v \mapsto x](v') = \begin{cases} 
  x & \text{if } v = v' \\
  \rho(v') & \text{otherwise}
\end{cases}
\]

If \( V \) is a set of variables, with \( V_C \subseteq V \) a subset of clock variables, then \( \rho[+\delta] \) denotes the application of a function \( \text{Env}_V \times \mathbb{R}^{\geq 0} \rightarrow \text{Env}_V \) that increases each clock in \( V \) by \( \delta \):

\[
\rho[+\delta](v) = \begin{cases} 
  \rho(v) + \delta & \text{if } v \in V_C \\
  \rho(v) & \text{otherwise}
\end{cases}
\]

The definition of XTG given here abstracts from two aspects of our concrete modeling language. The first, the format of the data modeling language, was already discussed. The second abstraction concerns the modeling of parallelism. The definition given below only allows the modeling of single graphs, which is sufficient for the purpose of this paper. The definition of a parallel composition for XTG is straightforward. It is similar to definitions of parallel compositions for timed automata found elsewhere. Basically, it allows the definition of sets of graphs that can synchronously pass values and operate on shared data.

**Definition 2.4 (XTG)**

An XTG is a tuple \( \langle V, \rho_0, L, l_0, I, E, U \rangle \), where

- \( V \) defines a set of variables where \( V_C \subseteq V \) defines the subset of clock variables, and \( V_P \subseteq V \) defines the subset of parameters with \( V_P \cap V_C = \emptyset \)
- \( \rho_0 : (V \setminus V_P) \rightarrow \text{Val} \) defines initial values for the non-parameter variables
- \( L \) is a set of location identifiers
- \( l_0 \in L \) identifies the initial location
- \( I : L \rightarrow \text{Bezpr}_V \) assigns an invariant to each location
- \( E \subseteq L \times \text{Bezpr}_V \times P(\text{Var} \times \text{Expr}_V) \times L \) is a set of edges. An edge is a tuple \( \langle l, g, u, e \rangle \), where \( l \) is the source location, \( g \) is a guard, \( u \) is a set of assignments and \( e \) is the destination location. Note that an assignment is defined as a set of pairs \( \{ v, e \} \), where \( v \) is a variable and \( e \) is a value expression, whose value is to be assigned to the variable.
- \( U : E \rightarrow \mathbb{R} \) identifies the subset of urgent edges

To be able to define the semantics of XTG in a convenient way, a function is defined that expresses in which situations edges of an XTG are enabled.

**Definition 2.5 (enabled edges)**

Given a set of variables \( V \), a function \( \text{trans} : \text{Expr}_V \times P(\text{Var} \times \text{Expr}_V) \times \text{Expr}_V \rightarrow \text{Env}_V \times \text{Env}_V \) is defined as in figure 2.
\[(\rho, \rho') \in \text{trans}(g, u, i) \iff \forall \nu \in \nu \cdot \\{ \begin{align*} &\rho'(\nu) = \rho(\nu) \quad \text{if } (\nu, -) \notin u \\ &\forall (\nu, e) \in u \cdot \rho'(\nu) = \nu[e](\rho) \quad \text{otherwise} \end{align*} \}\]

Fig. 2: The function \(\text{trans}\)

\[
\begin{align*}
(l, \rho) &\xrightarrow{u} (l', \rho') \iff \exists (l, g, u, l') \in E \cdot (\rho, \rho') \in \text{trans}(g, u, I(l')) \\
(l, \rho) &\xrightarrow{+\delta} (l, \rho[+\delta]) \iff \forall 0 \leq d \leq \delta \cdot \nu\nu[I(l)](\rho'[+d]) \text{ and} \\
&\forall 0 \leq d \leq \delta \cdot \exists (l, g, u, l') \in E \cdot \\
&U(l, g, u, l') \text{ and } (\rho, \rho') \in \text{trans}(g, u, I(l'))
\end{align*}
\]

where \(\delta > 0\) and \(S_0 = \{(l_0, \rho) \mid \forall \nu \in (V \setminus V_T) \cdot \rho(\nu) = \rho_0(\nu)\}\)

Fig. 3: XTG operational semantics

A pair of valuations \((\rho, \rho')\) is in \(\text{trans}(g, u, i)\) if the guard \(g\) is satisfied in \(\rho\), the invariant \(i\) is satisfied in \(\rho'\), and performing the assignments of \(u\) in \(\rho\) leads to \(\rho'\). Note that in case \(u\) contains conflicting assignments, the set \(\text{trans}(g, u, i)\) is empty, which means that edges containing conflicting assignments are disabled.

**Definition 2.6 (operational semantics of XTG)**

The operational semantics of an XTG \((V, \rho_0, L, l_0, I, E, U)\) is a timed structure \((L \times Env, S_0, T)\) where \(T\) is defined in figure 3 where \(\rightarrow\) is used to denote transitions in \(T\).

The first rule in the definition of \(T\) specifies that a discrete transition can be taken if an outgoing edge exists, that edge is enabled in the current valuation, and if the destination state reflects the update. The second rule states that a \(\delta\) time transition can be taken by increasing all clocks with a nonnegative value \(\delta\) if the invariant is not violated and if no urgent edges can become enabled 'along the way'.

### 2.2 Parametric specification of fair TCTL properties

Our property specification language is a temporal logic based on CTL (Computation Tree Logic) [7], TCTL (Timed CTL) [13], and fair TCTL [12]. The usage of these logics is strongly connected with the application of model checking verification. TCTL variants are real-time extensions of CTL. These extensions either augment temporal operators with time bounds, or use reset quantifiers. We use a similar approach by replacing the reset quantifier by assignments to property specification clocks and variables. In our TCTL variant \(z := 0.AF(p \land z \leq 10)\) expresses that \(p\) will always become true sometime within ten time-units. A secondary benefit from this approach is that we can use symbolic constants in our property formulae (as suggested in [10]). Consider for example, the TCTL formula \(AG(t := count.AF(count = t + 1))\) in which \(count\) is a system variable and \(t\) is property specification variable.

The core syntax defines two temporal operators, \(AU\) and \(EU\). The formula \((\phi_1 AU \phi_2)\) is satisfied in a state if for all computation paths starting from that state, there is a state along it which satisfies \(\phi_2\), and until that time \(\phi_1\) is satisfied. \((\phi_1 EU \phi_2)\) is satisfied if there is at least one such computation path. Derived operators are: \(EF\phi\) (There is a path on which there is state satisfying \(\phi\)), \(EG\phi\) (There is a path of which every state satisfies \(\phi\)), \(AF\phi\) (On all paths there is some state satisfying \(\phi\)), and \(AG\phi\) (On all paths every state satisfies \(\phi\)). There are two types atomic properties in our TCTL variant: boolean expressions over values of variables and clocks of both the system and the property, and location expressions. The latter are of the form \(g[l]\), and express the fact that the graph \(g\) of the system is currently at location \(l\).

Our property language allows the expression of fairness predicates. This allows the user to specify that unfair behaviours are not to be taken into account in the verification. A computation path is fair with respect to a set of fairness predicates, if each of these predicates is infinitely often satisfied along the path.

Finally, fair TCTL can be used for systems specified with parameters. The meaning of a fair TCTL specification interpretation on a parametric system is a set of parameter values for which the property holds. Parameters can also appear in property specifications. As an example, consider a property \(EFz > p\) where
is a variable of the system, and \( p \) is a TCTL parameter. This property determines the maximal value \( x \) can take.

**Definition 2.7 (fair TCTL core syntax)**
A fair TCTL specification for an XTG  
\( (V, \rho_0, L, l_0, I, E, U) \) is a tuple \( (PV, F, \phi) \), where
- \( PV \) is a set of property variables, having a subset \( PV_C \subseteq PV \) of clock variables and a subset of parameters \( PV_P \subseteq P \) such that \( PV_C \cap PV_P = \emptyset \).
- \( F \in \mathcal{P}(\text{Bezpr}_V) \) is a set of fairness predicates of the system’s variables, and
- \( \phi \) is a formula generated by the following syntax:
  \[
  \phi ::= g@l \mid b \mid \neg \phi \mid \phi \lor \phi \mid \phi \land \phi \\
  \phi AU \phi \mid \phi EV \phi \mid v := e. \phi
  \]

where
- \( g@l \) is a location expression in which \( g \) identifies a graph from the system and \( l \) a location in \( g \).
- \( b \in \text{Bezpr}_V \cup PV \) are boolean value expressions ranging over variables of the system as well as the property.
- \( v \in VP \) and \( e \in \text{Vexpr}_V \cup PV \), \( v := e \) thus denotes an update of a property variable.
- Any property specification variable \( v \) occurring in a boolean value expressions \( b \) have to be bounded by some earlier assignment operation \( v := e \).

The semantics of TCTL is not very different from those found in literature and is partly based on [12] and [23]. The main differences are the more expressive property update and the fact that properties are interpreted only over fair, divergent computation paths.

### 3 A model checking algorithm

#### 3.1 Introduction

The state space associated with the timed transition system for XTG is infinitely large, due to the reals in the value domain and the dense time domain of the XTG clocks. These states record the location of the control and the values of the variables and clocks. The algorithms for timed automata (and thus for timed structures) rely on partitioning the uncountable state space into finitely many regions and constructing a quotient called the region graph. States in the same region are in some sense equivalent, and the region graph is adequate for solving many problems (see e.g., [1]). The main problem with algorithms based on the region graph, is the size of the latter, which is exponential in the number of clocks and the length of the timing constraints. Bouajjani et Al. solve this problem in [6] by presenting an algorithm that - on the fly - minimalises a region graph under construction. The minimal graph is constructed using the algorithm presented in figure 4. The algorithm starts with a general transition system \( S = (S_0, s_0, \to) \) Let \( p \) be a partition of \( S \). Further it is assumed that \( p \) is a bisimulation. Let \( s \to X \) denote that \( s \to s' \) for some \( s' \in X \). A class \( X \) is defined as stable with respect to \( p \) if and only if the following holds

\[
\forall Y \in p. ([\exists x \in X, x \Rightarrow Y] \implies (\forall x \in X, x \Rightarrow Y)).
\]

A partition is a bisimulation if and only if every class of \( p \) is stable with respect to \( p \). The reduction of \( S \) according to \( p \) is the transition system \( S\delta(p) = (Acc(p), [s_0]_p, \to_p) \) where \( Acc(p) \) denotes the set of partition classes which contain at least one state accessible from \( s_0 \). \([s_0]_p \) denotes the class of \( p \) which contains the initial state \( s_0 \) and \( X \to_p Y \) if \( x \Rightarrow Y \) for some \( x \in X \). The algorithm computes starting from \( S \) and an initial partition \( p_0 \) the transition system \( S\delta(p) \) where \( p \) is the coarsest bisimulation compatible with \( p_0 \). The termination of the algorithm requires that the bisimulation \( \hat{p} \) have a finite number of classes. The

1. let \( p = p_0; \alpha = \{[s_0]_p\}; \sigma = \emptyset; \)
2. while \( \alpha \neq \hat{p} \) do
3. let \( X \in \alpha - \sigma; \)
4. let \( \alpha' = \text{split}(X, \rho); \)
5. if \( \alpha' = \{X\} \) then
6. let \( \sigma = \sigma \cup \{X\}; \alpha = \alpha \cup \text{post}_p(X) \)
7. else
8. let \( \alpha = \alpha - \{X\}; \)
9. if \( \exists Y \in \alpha'. \sigma_0 \in Y \) then
10. let \( \alpha = \alpha \cup \{Y\}; \)
11. end if
12. let \( \sigma = \sigma - \text{pre}_p(X); \)
13. let \( \rho = (\rho - \{X\}) \cup \alpha'; \)
14. end if
15. end while

**Fig. 4: Minimization Algorithm**

algorithm applies the following notation. The function \( \text{split} \) "splits" a class \( X \) of the partition \( p \) into a minimal set of subclasses that are stable with respect to \( p \). The function \( \text{post}_p(x) \) denotes the set of classes of \( p \) which contain at least one state directly accessible from a state of \( X \). Conversely the function \( \text{pre}_p(X) \) denotes the set of classes of \( p \) which contain at least one state from which a state of \( X \) is directly accessible. \( \rho \) is the current partition, \( \alpha \) is the set of classes of \( p \) which have been accessible from the class containing

0-7695-1435-9/02 $17.00 (c) 2002 IEEE
the initial state and $\sigma$ is the set of classes of $\rho$ which have been found stable with respect to $\rho$.

3.2 Our algorithm

This section presents a description of a partition refinement algorithm that checks whether or not some target location $l_d$ can be reached from the initial state $s_0$. The algorithm is derived from the abstract minimal model generation algorithm of [6]. It builds the reachable part of the partition of the state space induced by a greatest time-abstracting bisimulation that refines an initial partition. On-the-fly it checks if the verification problem can be decided. The algorithm starts from an initial partition and iteratively refines the partition by splitting unstable reachable classes, until all reachable classes are stable. In that case, the set of reachable classes coincides with the reachable subset of classes of the minimal partition. The initial partition is chosen in such a way that it separates states that originate from different locations, i.e. it is defined by the control graph of the system. As a consequence, the partitions occurring in the algorithm can be represented by region graphs, since its classes can be represented by regions. A region is combination of a control state and a zone, the latter corresponding to a set of valuations. The algorithm operates on-the-fly, which means that the partition is build while the state space is being explored.

Fig. 5 presents a simplified pseudo-code description of algorithm for reachability analysis. To keep the description of the algorithm simple, the bookkeeping of region graph edges is not made explicit. They are assumed to be added when appropriate. The following structures are used by the algorithm.

- $R_1$ the region of the initial partition that contains the initial state, $R_d$ is the region of the initial partition that is defined by the target location $l_d$ of the reachability analysis.
- $S$ is the set of nodes of which the edges have not yet been fully explored.
- $reachable(r)$ is an attribute that indicates whether or not a region $r$ is reachable from an initial region.
- $subnodes(r)$ denotes the set subnodes of $r$.

Starting from the initial region graph node $R_1$, a stable region graph is constructed in an iteration of three steps: (1) exploring a new edge (and possibly a new location), (2) stabilizing the region graph, and (3) use the new information to possibly decide values of nodes. The state space is thus explored by iteratively incorporating new X1G edges. When incorporating a new edge, the algorithm will often encounter locations that were not yet visited. For such locations an initial region graph node is created. These newly created nodes are immediately split to create subnodes that respect the guards of the outgoing edges of this location.

These splittings may cause stable predecessors of this node to become unstable, because states corresponding to these predecessor nodes may now have successors in different nodes. The stabilize function takes care of this. It splits a region that is unstable with respect to an edge to another region into smaller regions that again stable. The stabilize operation is recursive because stabilizing a node may again cause predecessor nodes to become unstable. In this manner splits are propagated backwards through the graph. The key operation in the stabilize function is propagate. Suppose there are two nodes $r$ and $r'$, and an edge from $r$ to $r'$. Furthermore suppose that $r'$ is split into smaller subnodes. Then propagate splits $r$ into subnodes that are stable with respect to the edges to subnodes of $r'$. One could say it ‘propagates’ the splitting of $r'$ back to $r$.

The backward propagation technique is combined with a reachability analysis mechanism, to avoid unnecessary splitting of unreachable nodes. The stabilize operation keeps track of the reachability properties and avoids splitting unreachable nodes.

It is obvious that the algorithm relies on two operations:

- $propagate(r, r')$: split all subnodes of $r$ into nodes that are stable with respect to edges to $r'$.
- $split(r, c)$: split all subnodes of $r$ into nodes that respect $c$

These are the two operations that our symbolic state space will be expected to provide. This is also true for applications to more complex verification problems. Indeed, for more complex verification problems, the algorithm is very similar. The main difference is that there is a more complex mechanism for on-the-fly evaluating whether or not the property can be decided on the current partition. Thus, by constructing an state space representation that allows efficient implementation of the two operations, we open the way for a range of verification solutions based on partition refinement. Due to space limitations in this paper we restrict ourselves to the discussion of the $split$ operation. A full description of the algorithm is presented in [21].

3.3 Splitting trees

A partition refinement technique is only useful if partitions can be represented symbolically, i.e. as symbolic state spaces. We will describe how symbolic state spaces resulting from a partition refinement algorithm can be represented efficiently. Traditionally, in symbolic model checking canonical representations
1. \( \text{reachable}(R_i) = \text{true} \); \( S = \{ R_i \} \)
2. while \( S \neq \emptyset \) do
3.   let \( r \in S \)
4.   if \( r \) is a split node then
5.     \( S = S \cup \{ r' \in \text{subnodes}(r) \mid \text{reachable}(r') \} \setminus \{ r \} \)
6.   else
7.     if there exists and enabled, unexplored edge \( e \) from \( r \) then
8.       if the destination location \( l \) of \( e \) was not yet visited then
9.         create new node \( nr \) for \( l \); \( \text{reachable}(nr) = \text{false} \); \( S = S \cup \{ nr \} \)
10.      for all outgoing edges \( e' \) of the location \( l \) do
11.         split(nr, guard), where \( \text{guard} \) denotes the guard of \( e' \)
12.     end for
13.     stabilize(r, nr)
14.     if there exists a region \( r \subseteq R_d \) s.t. \( \text{reachable}(r) \) then
15.       return true
16.     end if
17.   else
18.     \( S = S \setminus \{ r \} \)
19.   end if
20. end while
21. return false
22. stabilize(r, r');
23. propagate(r, r')
24. for all nodes \( rr \) having an edge to \( r \) do
25.   stabilize(\( rr \), r)
26. end for
27. for all \( sr \in \text{subnodes}(r) \) do
28.   if \( sr \subseteq R_i \) or there exists a node \( rr \) with an edge to \( sr \) s.t. \( \text{reachable}(rr) \) then
29.     \( \text{reachable}(sr) = \text{true} \)
30.   end if
31. end for
32. end for

Fig. 5: Simplified reachability analysis algorithm

have been used. However, the time and data resources needed for maintaining such canonical representations have great impact on the performance of the model checking techniques. Most model checking approaches require several operations on regions, which justifies the burden of using canonical representations, because all these operations are performed relatively efficient on such representations. When applying a partition refinement technique, this is not a natural approach because the required operations on the state space are very limited and specific. We already say that only two basic operations are required, namely split and propagate. In this case, there is no advantage in using canonical structures like DBM’s. As we will show, these operations can be implemented surprisingly simple on state spaces represented by a dedicated, more efficient representation.

Our model checking technique operates on set of states that can be seen as tuples \((l, Z)\), where \( l \) is a location and \( Z \) is a set of valuations for the system’s variables. Essentially, what is needed is a representation for zones. Zones occurring in symbolic algorithms are defined as sets of constraints. Canonical representations like Difference Bound Matrices, keep a canonical model for each zone. We took a different approach, which is inspired by the fact that zones are generated by a partition refinement algorithm. In the partition refinement approach zones are the consequence of repeated splitting. It therefore seems natural to represent zones as leaves of a so-called splitting tree. A splitting tree is a binary tree that records a history of splittings. Internal nodes of the tree are labeled with constraints, which represent the splitting of a zone into two subzones, one satisfying the constraint and one dissatisfying the constraint. Leaf nodes define zones that are relevant for the symbolic state space. The set of constraints defining the zone associated with a region is obtained by collecting all constraints while
traversing the tree upwards from the node, negating constraints when appropriate.

This leads to a forest structure in which for every location, a splitting tree is kept which holds for that location. The set of leaf nodes of the tree associated with a location represents the complete set of zones that are relevant for that location. Thus, leaf nodes correspond to region graph nodes of the region graph, where the location is identified by the particular tree in the forest and the zone is defined by the splitting history.

Splitting trees can be used to model zones generated by constraints on any type of data. Here, we will focus on zones over the domain of reals, since these are of primary importance because these are the result of clock variables and real variables in the system. We demand that constraints (on reals) in a splitting tree can only be single linear equations. By doing so it is ensured that the leaf nodes of the splitting trees always represent convex zones. The benefit of that choice will become clear in the next section, when discussing the operations on the splitting trees.

There are several advantages of using splitting trees to represent a symbolic state space. As will be shown below, it allows efficient implementation of the two basic operations needed by the partition refinement algorithm. Furthermore, the creation of new regions is a computationally simple operation, which is a clear difference from canonical representations, where each region has to be brought into a canonical form. Finally, it has proved to be relatively efficient with respect to the amount of space it requires. Splitting trees were introduced in [22] also also applied in [20], but these approaches use splitting trees solely for the purpose of propagation, not for state space representation. The remainder of this section demonstrates the implementation of the two basic operations, propagation and splitting, on state spaces represented by splitting trees.

3.4 Splitting and reduction

Given a node $n$ and a constraint $c$, the split operation minimally splits $n$ into subnodes that respect $c$. For regions represented by the splitting tree representation, the splitting of a region is in fact a trivial operation. Given a region represented by a leaf node of some splitting tree, splitting this region means that the node becomes an internal node labeled with the new constraint and that two new child nodes are added. These latter nodes represent the regions that result from the splitting.

However, what is needed is a mechanism to avoid the useless splitting of nodes. If a zone respects a constraint, then splitting it with the constraint is not useful. The mechanism of avoiding such useless splittings, which we refer to as reduction, is applied whenever a region split is considered. If reduction would not be applied, the algorithm would in most cases keep on splitting eternally. Given a zone $Z$ defined by a set of constraints $C$ and a new constraint $c$, the reduction procedure integrates two tests, one checking whether or not $\forall C \Rightarrow c$, and another one checking $\exists C \Rightarrow \neg c$. If either condition is true, then no splitting needs to be performed. One reason for restricting ourselves to convex nodes lies in the fact that it allows us to efficiently solve the reduction problem.

The reduction algorithm is a critical part of our model checking approach. In our model checking implementations it proved to be the most time consuming operation by far. Furthermore, reduction is the single operation that limits the type of constraints that can be used to define regions. This limitation is directly related to the type of verification problems that can be handled by the model checker, in particular the complexity of the guards and updates occurring in an XTG model. Thus, the reduction algorithm is a central aspect of our model checking approach, since it largely determines the efficiency of a model checker as well as the complexity of the models it can handle. As one would expect, a higher level of complexity tends to come with a lower efficiency. To further explore this relation, we distinguish three categories of constraints:

1. simple constraints: $x - y \sim c$ and $s \sim c$ where $x$ and $y$ are variables and $c$ is an integer constant.
2. linear constraints: $ax + by + \ldots \sim c$, where $a$, $b$, and $c$ are integer constants, $x$ and $y$ are variables.
3. non-linear constraints

For the first type of constraints, efficient representations can be found. Efficient model checkers like Uppaal [16] and Kronos [8] rely on the limitation to simple constraints. For linear constraints things are getting considerably more complex, only few model checkers [3] implementing it. Finally, for non-linear constraints, model checking becomes mostly infeasible, although for some limited problems, solutions have been found [11].

Reduction on regions defined by simple constraints can be performed by a variation of a shortest path algorithm. A convex zone defined by a set of simple constraints can conceptually be represented by a graph in which the vertices represent variables and edges represent constraints on these variables: the presence of an edge $x \rightarrow y$ means that $x - y < 3$ is one of the constraints defining the zone. An additional node $0'$ is introduced to represent constraints on single variables: an edge $x \rightarrow 0'$ represents the fact that $x < 3$. Given a new constraint $x - y < c$, the reduction procedure checks if there is a path from $x$ to $y$ with an accumulated weight less than $c$ (thus $\exists C \Rightarrow x - y < c$), or
a path from $y$ to $x$ with an accumulated weight less than $-c$ ($\lambda C \Rightarrow \neg(x-y < c)$). This approach is easily extended to also deal with non-strict constraints.

For linear constraints the reduction problem can be formulated as a linear programming problem. Again let $C$ be the set of constraints that define a zone, and $c$ a new constraint, then the reduction operation can be done by checking

1. whether or not $\lambda (C \cup c)$ is feasible, and
2. whether or not $\lambda (C \cup \neg c)$ is feasible

If for both are feasible, the new constraint $c$ represents a true split. Both problems can obviously be solved by standard linear programming algorithms. We built our own dedicated linear programming package, because an LP solver was needed that efficiently solves large amounts of relatively small LP problems. This requirement does not match very well to existing LP solvers, since these tend to focus on efficiently solving huge problems.

4 Practical results

We have extended the approach described in section 3 towards the parametric verification of fair TCTL property on systems specified in a modeling language called XTG. Fair TCTL is an extension of TCTL [13] in the spirit of [12]. Parametric verification allows one to specify parameters both in system and parameters, the result of a parametric verification being a set of constraints on the parameters defining the set of parameter valuations that cause the property to be satisfied. Instead of operating on a region graph, the extended algorithm operates on a labeled region graph. A labeled region graph is a product structure of a graph representation of the property and a region graph. The decomposition of properties into graph representations is based on fixed point characterisations of TCTL formulae. The product structure approach is partly based approaches followed in [20] and [12]. Details can be found in the forthcoming [21].

The type of constraints that may occur in the exploration determines the type of systems and verification problems that can be handled by the model checker. If a system specification contains only simple constraints on clocks then only simple constraints will occur in the exploration. Likewise, if non-simple, linear constraints occur in a system specification, linear constraints will occur in the exploration. Furthermore, if besides clocks also real variables are allowed in specifications, and these variables occur together with clocks in the specification, also non-simple constraints will occur. As a consequence, most parametric real-time verification problems will need a model checker that is based on linear constraints, since parameters will often be compared with clocks. Finally, being able to deal with linear constraints also allows the verification of a limited set of hybrid systems.

We have build two similar model checking tools, one with the reduction routine for simple constraints and one with the reduction routine for linear constraints. The first one (called PMC) is able to efficiently deal with XTG specifications in which only simple constraints are used and in which discrete variables (including parameters) cannot be mixed with clock variables. The second one (called LPMC) is able to deal with systems specified using linear equations in updates and constraints. Furthermore it allows full parametric verification, and the verification of limited set of hybrid systems. Besides real variables, both tools allow the definition of integer and enumerated variables. Furthermore both tools are able to generate diagnostic traces.

When comparing to other tools, one would have to compare PMC with Uppaal [16] and compare LPMC with HyTech [3]. In [17] we showed already that an early version of PMC performed remarkably well compared to Uppaal. Comparing HyTech to LPMC shows that LPMC performs significantly better than HyTech ([4]).

5 Conclusions and further work

We presented a new model checking approach based on partition refinement. This approach includes a novel technique for representing symbolic state spaces called splitting trees. Unlike canonical representations like DBM’s, splitting trees are not generally applicable in real-time symbolic model checking. The splitting tree representation is dedicated to partition refinement approaches, and therefore relatively efficient. The idea of using a splitting tree is independent of the kind of data that is represented. This makes the incorporation of other data types in the input language of our model checker relatively easily. It should not be hard to extend our model checker to also deal with other data types. All that is required is the definition of propagation and reduction procedures for the data type.

The basic idea behind our approach is quite simple. In [22] it was already shown that splitting trees are very useful for implementing the propagation operation. However, instead of using a standard canonical state space representation to represent the regions, we extended the usage of the splitting trees to also act as the data structure that defines the region. This works because of the fact that in a partition refinement approach, besides propagation, only one other operation is required, which involves the checking of feasibility of
regions. This operation can be implemented relatively efficient on regions represented by sets of constraints.

Based on the ideas presented in this article we have build two model checkers that have proved to be strong both in terms of performance as well as expressiveness of the input languages. Especially LPMC, allows both in terms of performance as well as expressiveness, to build two model checkers that have proved to be strong compared to existing model checking tools.

However, we still see much room for improvement. As was mentioned before, the reduction algorithm is the central part of our model checking approach. We are currently further investigating approximate reduction strategies.

References


