Local Model Checking for Real-Time Systems
(Extended Abstract)*

Oleg V. Sokolsky Scott A. Smolka

Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794-4400
{oleg,sas}@cs.sunysb.edu

Abstract. We present a local algorithm for model checking in a real-time extension of the modal mu-calculus. As such, the whole state space of the real-time system under investigation need not be explored, but rather only that portion necessary to determine the truthhood of the logical formula. To the best of our knowledge, this is the first local algorithm for the verification of real-time systems to appear in the literature.

Like most algorithms dealing with real-time systems, we work with a finite quotient of the inherently infinite state space. For maximal efficiency, we obtain, on-the-fly, a quotient that is *as coarse as possible* in the following sense: refinements of the quotient are carried out only when necessary to satisfy *clock constraints* appearing in the logical formula or timed automaton used to represent the system under investigation. In this sense, our data structures are optimal with respect to the given formula and automaton.

1 Introduction

The Concurrency Factory [CGL+94] is a joint project between the State University of New York at Stony Brook and North Carolina State University to develop an integrated toolset for the specification, verification, and implementation of concurrent and distributed systems. Like the Concurrency Workbench [CPS93], the Factory employs bisimulation, preorder, and model checking as its main avenues of analysis.

A major underlying goal of the project is that the Factory be suitable for industrial application. One manner in which we are striving to achieve such applicability is through the use of *local*, or *on-the-fly*, verification techniques. In a local approach to verification, only the portion of the state space necessary to determine the outcome of the verification procedure is explored. As such, local techniques provide a powerful heuristic for dealing with complex (i.e. large state space) specifications.

The Factory is currently equipped with a local model checker for the modal mu-calculus; an incremental model checker has also been implemented [SS94]. *Model checking* [CE81, CES86] is the problem of verifying whether a system possesses a property specified by a formula in some temporal logic (in other words, provides a model for

* Research supported in part by NSF Grants CCR-9120995 and CCR-9208585, and AFOSR Grant F49620-93-1-0250.

the formula). The *modal mu-calculus* [Koz83] is a highly expressive logic that can be used to specify safety and liveness properties of concurrent systems represented as labeled transition systems (LTSs). Besides the standard logical connectives, the modal mu-calculus consists of dual modal operators $\Box$ (necessarily) and $\Diamond$ (possibly), and dual fixed-point operators μ (least fixed point) and ν (greatest fixed point). This logic is often referred to as L_μ, and so it shall be here. Our local model checker for L_μ is a variant of the graph-based algorithm of Anderson [And94], which is the most efficient of the local model checking algorithms proposed to date.

Industrial applications also often require support of *real-time* specifications of both the system under development and the properties the system is required to comply with. Real-time specifications, in which the time domain is taken to be the nonnegative real numbers, make it possible to catch subtle timing errors in the system behavior which may otherwise be abstracted away in a discrete-time model. The price to be paid is that a dense time domain adds a level of complexity to the verification process, as the state space of such systems is inherently infinite (and uncountable): any approach to *automatic* verification will necessitate the construction of some *finite quotient* of the state space [AD94].

This current paper is concerned with extending local model checking to real-time specifications, in particular, *local model checking in a real-time extension of the modal mu-calculus*. Our focus here is on a real-time extension of the *alternation-free* fragment of the modal mu-calculus [EL86] which, intuitively, means that the "level" of mutually recursive greatest and least fixed-point operators is one. This fragment, referred to as L_{μ_1}, is still powerful enough to express most of the properties of interest. For example, CTL [CE81] has a uniform encoding into L_{μ_1}.

Our main result is a local algorithm for model checking in a real-time extension of the alternation-free modal mu-calculus. The principal innovations of our algorithm, which we call *TMC* (Timed Model Checking), are the following:

- *TMC* is, to our knowledge, the first *local* model checking algorithm to be proposed for the verification of real-time systems. Thus, like its counterparts for untimed systems [SW91, Cle90, Lar92, And94, VLAP94], the whole state space need not be explored, but rather only that portion necessary to determine the truthhood of the formula.
- Also, to the best of our knowledge, we present the first true extension of the modal mu-calculus to real-time systems. Our logic, L_μ^t, supports all of the original operators of L_μ as well as the two new *time modalities* of [HLW91]: necessity and possibility of *time successors*. Moreover, we achieve a clear separation of the time-dependent aspects of the semantics of our logic from the untimed ones. This will allow us to reuse a significant portion of the code of the Concurrency Factory's local model checker for the modal mu-calculus when implementing the local model checker for L_μ^t.
- Like most algorithms dealing with real-time systems [Dil89, ACD93, ACD+92], we work with a finite quotient of the state space as the state space itself is inherently infinite. For maximal efficiency, we obtain a quotient that is *as coarse as possible* in the following sense: refinements of the quotient are carried out only when necessary to satisfy *clock constraints* appearing in the L_μ^t formula or timed automaton [AD94, HNSY94] used to represent the system under investigation. In this sense, our data structures are optimal with respect to the given formula and

automaton.

We approach the model checking problem by constructing a data structure representing the "product" of the given logical formula and the transition system induced by the given timed automaton. Each node of this *region product graph* (RPG) represents the value of a logical variable for some set of *timed states*, or *region*. A similar, albeit untimed, product construction is employed in [SS94, And94, EJS93] and in the automata-theoretic approach of [VW86, BVW94]. The RPG is constructed on-the-fly using methods similar to [BFH$^+$92, ACH$^+$92]. However, rather than viewing regions as sets of states, we treat them as sets of clock constraints and reason directly in these terms. The RPG is explored in a depth-first manner, until nodes with a known value are found. After each step of the RPG construction, partitioning (or splitting) of nodes may be necessary to achieve stability with respect to the relevant clock constraints. It is worth noting that if the real-time system is represented as the *composition* of a collection of timed automata, then the global automaton is also constructed on-the-fly.

In terms of related work, a number of non-local algorithms for model checking real-time logics have appeared in the literature, including [ACD93, Alu91, HNSY94]. The algorithm of [HNSY94] is particularly relevant as it supports a real-time mu-calculus, albeit with a different set of modalities than those in L_μ^t. We believe that our algorithm — while not as abstract as some of these approaches, which work with a highly symbolic representation of the state space — is easier to understand and exhibits the efficiency that is characteristic of local verification techniques.

Our method of keeping track of the history of RPG node splits (see Section 5) is motivated by the timed transition system minimization algorithm of [YL93]. The purpose of our algorithm, however, is quite different from theirs: we attempt to avoid constructing the entire state space quotient, even the minimal one. Thus, many of the techniques of [YL93] aimed at system minimization are not applicable in our setting.

There is also a large body of work on discrete-time logics. Of particular interest is the discrete-time mu-calculus of [Eme91], where time is equated with the number of iterations required to reach a fixed point of a given mu-calculus formula.

2 Timed Automata

To model real-time systems, we use a version of timed graphs [AD94] called timed safety automata in [HNSY94]. Before defining these automata, some definitions concerning clocks and regions are in order. A *clock* is simply a real-valued variable drawn from the countably infinite set $\mathcal{C}$. A *clock constraint* $\mathbf{c}$ is an expression of the form $x \, R \, c$ or $x + c \, R \, y + d$, where x, y are clocks, c, d are integer constants, and R is taken from $\{\leq, \geq, <, >\}$. A *clock assignment* π is a point in $\mathbf{R}^n$. If π is a clock assignment, $\pi + d$ for some $d \in \mathbf{R}$ stands for the clock assignment obtained by adding d to the value of every clock. Finally, let $\xi \subseteq \mathcal{C}$ be a set of clocks. Then $\pi[\xi := 0]$ is the clock assignment obtained from π by setting every clock in ξ to 0.

A *region* is a subset of $\mathbf{R}^n$ formed by a set of clock constraints. We will use ρ, possibly primed or subscripted, to range over regions, and Σ to denote the set of all regions. A region ρ is *past-closed* if whenever $\pi \in \rho$, for every $d \in \mathbf{R}$, $\pi - d \in \rho$.

A *timed safety automaton* (TSA) is a 6-tuple $\langle \mathcal{L}, Act, \xi, \longrightarrow, l_0, L \rangle$, where

- $\mathcal{L}$ is a finite set of *locations*.
- *Act* is the set of *actions*.
- ξ is a finite set of real-value *clocks*.
- $\longrightarrow \subseteq \mathcal{L} \times Act \times \Sigma \times 2^{\xi} \times \mathcal{L}$ is the *transition relation*, where every transition is labeled by an action, a region, and a set of clocks.
- $l_0 \in \mathcal{L}$ is the *start location*.
- $L : \mathcal{L} \rightarrow \Sigma$ is the the function assigning a past-closed region, called a *location invariant*, to every location.

Intuitively, a TSA operates by taking transitions from location to location. Executing a transition takes no time. If no transitions are taken, time progresses by incrementing every clock by an arbitrary real number. The intent of the labels on locations and transitions is the following: The automaton can stay in a location l as long as $L(l)$ is satisfied. If there is always a way for the automaton to leave the location before its invariant is violated, the automaton is called *non-Zeno*. We restrict our attention to non-Zeno automata, as it was shown in [HNSY94] that every automaton can be turned into an equivalent non-Zeno one by strengthening its invariants. Regions labeling the transitions are *enabling conditions*, i.e. a transition can only be taken if the constraints defining its enabling condition are satisfied. Also, when a transition occurs, all clocks in the set of clocks labeling it are reset to 0.

The semantics for TSAs are given in terms of labeled transition systems, which are often called *dense* due to the uncountability of their state sets. Formally, every TSA induces a dense transition system $\mathcal{D} = \langle \mathcal{S}, Act, \longrightarrow, \langle l_0, \pi_0 \rangle \rangle$, where $\mathcal{S}$ is a set of *timed states*, i.e. pairs of the form $\langle l, \pi \rangle$, $l \in \mathcal{L}$, $\pi \in \mathbf{R}^n$; $\langle l_0, \pi_0 \rangle$ is the timed start state with π_0 assigning 0 to every clock; $\longrightarrow \subseteq \mathcal{S} \times Act \cup \{\epsilon\} \times \mathcal{S}$ with $\epsilon \notin Act$ being a distinguished action name, is the transition relation defined by the following rules:

- $\langle l, \pi \rangle \xrightarrow{a} \langle l', \pi[\eta := 0] \rangle$, if $l \xrightarrow{a,\rho,\eta} l'$, $\pi \in \rho$, and $\pi \in L(l), \pi[\eta := 0] \in L(l')$.
- $\langle l, \pi \rangle \xrightarrow{\epsilon} \langle l, \pi + d \rangle$ for every $d \in \mathbf{R}$ such that $\pi, \pi + d \in L(l)$.

In the first rule, the automaton moves from location l to l' via action a. Time does not progress but all clocks in η, $\eta \subseteq \xi$, are reset. Moreover, it must be the case that $L(l)$ and ρ are satisfied by π and $L(l')$ is satisfied by $\pi[\eta := 0]$. Timed state $\langle l', \pi[\eta := 0] \rangle$ is said to be a *transition successor* of $\langle l, \pi \rangle$. In the second rule, time progresses (by d) and the automaton remains in location l, provided that π and $\pi + d$ satisfy $L(l)$. Timed state $\langle l, \pi + d \rangle$ is said to be a *time successor* of $\langle l, \pi \rangle$.

3 Timed Modal Mu-Calculus

We will specify properties of timed automata using the logic L^t_μ, a real-time extension of the modal mu-calculus. Our logic is inspired by both Timed HML [Wan91, HLW91] and the timed mu-calculus T_μ of Henzinger *et al.* [HNSY94]. As in [CS93], a formula of L^t_μ is a set of *blocks* of mutually recursive equations of the form $X = \phi$, with operator *min* or *max* applied to each block. These operators are understood respectively as the least and greatest fixed points of the set of equations. A variable X is said to be defined in a block B if it appears on the left-hand side of an equation in B. We will restrict our attention to *closed* formulas, where all variables are defined.

The right-hand sides of equations in a block, referred to as *basic formulas*, may contain variables defined in other blocks. We say that block B_1 depends on B_2 if a variable defined in B_2 is used in B_1. We further restrict our attention to formulas whose dependency relations are acyclic. This ensures that no *alternating fixed points* [EL86] can occur. The syntax of basic formulas is given by the following grammar:

$$\phi ::= A \mid X \mid A_t \mid \phi \vee \phi \mid \phi \wedge \phi \mid [a]\phi \mid \langle a \rangle \phi \mid \exists \phi \mid \forall \phi \mid x.\phi$$

Here, a is an action, x is a clock, and A_t is a clock constraint as defined above. Atomic propositions A and variables X are taken from the countably infinite sets $\mathcal{A}$ and Var.

Operators $[a]$, $\langle a \rangle$, $\vee$ and $\wedge$, as well as atomic propositions can be found in the propositional modal mu-calculus [Koz83]. The remaining operators form our real-time extension of the logic. The first two of the new operators come from T_μ [HNSY94]: A_t asserts a relationship between clock values in a state and $x.\phi$ is a *reset* operation. The remaining operators, $\exists$ and $\forall$, allow us to reason about time successors of a timed state. They appeared in [HLW91].

Modal operators $\langle a \rangle$ and $[a]$ are used to reason about transition successors of a given timed state: $\langle a \rangle$ provides existential quantification over the state's transition successors. That is, $\langle a \rangle \phi$ is true in timed state s if from s the automaton can *immediately* (i.e. without letting time progress) take an a-transition and reach a timed state where ϕ holds. Dually, $[a]$ serves as universal quantification. In a similar fashion, $\exists$ and $\forall$ provide quantification over time successors of a timed state: $\exists \phi$ is true in timed state s if there exists a time increment, i.e. a real value, that when added to each clock in s, yields a timed state in which ϕ holds.

An example L_μ^t formula is as follows.

$$\begin{array}{ll} max & \{\, Y = \forall[-]Y \wedge \forall[a]\, z.X \quad \} \\ min & \{\, X = \exists\langle b \rangle z \leq 5 \vee (\forall[-]X \wedge \exists\langle - \rangle tt) \,\} \end{array}$$

The dash '$-$' in modalities stands for "any action" and is a syntactic abbreviation for a boolean expression over ordinary modalities. The formula states that after performing action a, it is always possible to engage in action b within 5 units of time.

Given a TSA $T = \langle \mathcal{L}, Act, \xi, \longrightarrow, l_0, L \rangle$, basic formulas are interpreted with respect to the dense transition system $\mathcal{L}_T = \langle \mathcal{S}, Act, \longrightarrow, \langle l_0, \pi_0 \rangle \rangle$ induced by T', which is T with ξ extended by all clocks mentioned in ϕ,[2] a *valuation* mapping $\mathcal{V} : \mathcal{A} \to \mathcal{P}(\mathcal{S})$, and an *environment* $e : Var \to \mathcal{P}(\mathcal{S})$. Intuitively, a valuation mapping (environment) maps each atomic proposition (variable) to the set of timed states in which it holds. The valuation mapping is assumed to be untimed in the sense that, for all timed states $\langle l, \pi \rangle$ corresponding to the same location l, each proposition A has the same value. For a fixed environment e, the meaning of basic formulas is given by the semantical function $[\![\cdot]\!]e : \Phi \to \mathcal{P}(\mathcal{S})$, defined in Figure 1.

Blocks are understood semantically as functions from environments to environments. Let a block B contain a set of equations E with variables $X_1, \ldots, X_n$ defined as left-hand sides. Let $\overline{S} = \langle S_1, \ldots, S_n \rangle \in (2^{\mathcal{S}})^n$ and let $e_{\overline{S}} = e[X_1 \mapsto S_1, \ldots, X_n \mapsto S_n]$. Then the function

$$f_E^e(\overline{S}) = \langle [\![\Phi_1]\!]e_{\overline{S}}, \ldots, [\![\Phi_n]\!]e_{\overline{S}} \rangle$$

[2] For every TSA and L_μ^t formula, we assume that their sets of clocks are disjoint.

$$\begin{aligned}
&[\![A]\!]e = \mathcal{V}(A)\\
&[\![X]\!]e = e(X)\\
&[\![A_t]\!]e = \{\langle l, \pi\rangle \mid \pi \in A_t \wedge L(l)\}\\
&[\![\phi_1 \vee \phi_2]\!]e = [\![\phi_1]\!]e \cup [\![\phi_2]\!]e\\
&[\![\phi_1 \wedge \phi_2]\!]e = [\![\phi_1]\!]e \cap [\![\phi_2]\!]e\\
&[\![\langle a\rangle\phi]\!]e = \{\langle l, \pi\rangle \mid \exists\langle l', \pi'\rangle.\ \langle l, \pi\rangle \xrightarrow{a} \langle l', \pi'\rangle \wedge \langle l', \pi'\rangle \in [\![\phi]\!]e\}\\
&[\![[a]\phi]\!]e = \{\langle l, \pi\rangle \mid \forall\langle l', \pi'\rangle.\ \langle l, \pi\rangle \xrightarrow{a} \langle l', \pi'\rangle \Rightarrow \langle l', \pi'\rangle \in [\![\phi]\!]e\}\\
&[\![\exists\phi]\!]e = \{\langle l, \pi\rangle \mid \exists d \geq 0.\ \langle l, \pi\rangle \xrightarrow{\epsilon} \langle l, \pi'\rangle \wedge \langle l, \pi'\rangle \in [\![\phi]\!]e\}\\
&[\![\forall\phi]\!]e = \{\langle l, \pi\rangle \mid \forall d \geq 0.\ \langle l, \pi\rangle \xrightarrow{\epsilon} \langle l, \pi'\rangle \Rightarrow \langle l, \pi'\rangle \in [\![\phi]\!]e\}\\
&[\![x.\phi]\!]e = \{\langle l, \pi\rangle \mid \langle l, \pi[x := 0]\rangle \in [\![\phi]\!]\}
\end{aligned}$$

Fig. 1. Semantics of basic formulas.

defined on the lattice of tuples of sets of timed states ordered by point-wise set inclusion is monotonic. By the Tarski-Knaster fixed-point theorem, f_E^e has both greatest and least fixed points given by:

$$\nu f_E^e = \bigcup\{\overline{S} \mid \overline{S} \subseteq f_E^e(\overline{S})\}$$
$$\mu f_E^e = \bigcap\{\overline{S} \mid f_E^e(\overline{S}) \subseteq \overline{S}\}$$

Blocks can now be interpreted in the following fashion:

$$[\![maxE]\!]e = e_{\nu f_E^e}$$
$$[\![minE]\!]e = e_{\mu f_E^e}$$

The meaning $[\![\mathcal{B}]\!]e$ of the formula $\mathcal{B}$ containing blocks $B_1, \ldots, B_m$, topologically sorted by the dependency relation, can be computed through the sequence of environments $e_1 = [\![B_1]\!]e$, …, $e_m = [\![B_m]\!]e_{m-1}$, with $[\![\mathcal{B}]\!]e = e_m$. Due to the acyclicity restriction on the dependency relation, we are ensured that $[\![\mathcal{B}]\!]e_m = e_m$. If $\mathcal{B}$ is closed, then for every two environments e and e', we have $[\![\mathcal{B}]\!]e = [\![\mathcal{B}]\!]e'$. In this case, we omit the reference to the environment in $[\![\mathcal{B}]\!]$.

The *model checking problem* for L_μ^t can now be defined: Given a TSA A with start location l_0, and an L_μ^t formula $\mathcal{B}$ with a designated variable X_0 (called henceforth the *main* variable) defined within it, determine whether $\langle l_0, \pi_0\rangle \in [\![X_0]\!][\![\mathcal{B}]\!]$.

4 Region Product Graphs and Fixed Point Computation

This section introduces region product graphs, the main data structure employed by our local model checking algorithm for L_μ^t, and shows how these graphs can be used to iteratively carry out the requisite fixed point computation. We first define a graph representation of L_μ^t formulas called the *formula graph*. For this purpose, we assume, as in [CS93], that the basic formulas appearing on equation right-hand sides are *simple*; i.e. they are atomic propositions or clock constraints, or formed by the application of exactly one operator to variables. The vertices of a formula graph are the variables of the formula not defined by a reset operator, and there is an edge from X to X' if X is defined in terms of X'. A vertex is labeled by the operator defining the variable in question and is written $X(op)$, while an edge is labeled by the set of clocks reset in going from the source to the destination vertex.

As a precursor to our definition of region product graph, we introduce the notion of a timed product graph, the product of a formula graph and dense transition system induced by a TSA, extended by the clocks of the formula. Although, the timed product graph data structure is infinite and therefore unsuitable for algorithmic purposes, it will prove convenient for reasoning about region product graphs.

Let $\mathcal{D} = \langle \mathcal{S}, Act, \longrightarrow, \langle l_0, \pi_0 \rangle \rangle$ be a dense transition system and $\mathcal{B}$ an L_μ^t formula. Then the *timed product graph* of $\mathcal{D}$ and $\mathcal{B}$ is the directed graph with set of vertices $\{\langle X, l, \pi \rangle \mid X \in Var, \langle l, \pi \rangle \in \mathcal{S}\}$ and set of edges given by:

- if $X(\vee) \xrightarrow{\xi} X'$ or $X(\wedge) \xrightarrow{\xi} X'$, then for every $\langle l, \pi \rangle \in \mathcal{S}$, $\langle X, l, \pi \rangle \longrightarrow \langle X', l, \pi[\xi := 0] \rangle$.
- if $X(\langle a \rangle) \xrightarrow{\xi} X'$ or $X([a]) \xrightarrow{\xi} X'$, $\langle l, \pi \rangle \xrightarrow{a} \langle l', \pi \rangle$, then $\langle X, l, \pi \rangle \longrightarrow \langle X', l', \pi[\xi := 0] \rangle$.
- if $X(\exists) \xrightarrow{\xi} X'$ or $X(\forall) \xrightarrow{\xi} X'$, $\langle l, \pi \rangle \xrightarrow{\epsilon} \langle l, \pi' \rangle$, then $\langle X, l, \pi \rangle \longrightarrow \langle X', l, \pi'[\xi := 0] \rangle$.

Thus there is an edge from $\langle X, l, \pi \rangle$ to $\langle X', l', \pi' \rangle$ if the value of X in timed state $\langle l, \pi \rangle$ depends on the value of X' in timed state $\langle l', \pi' \rangle$. If $\vee$, $\langle a \rangle$, or $\exists$ are used to define X, the vertex $\langle X, l, \pi \rangle$ is called an $\vee$-*node*; otherwise, it is called an $\wedge$-*node*.

A *region product graph* (RPG) is a finite quotient of a timed product graph whose nodes are *sets* of timed product graph nodes with the same variable and location components and whose time components form a region. For variable X, location l, and region ρ, we denote such an RPG node as $\langle X, l, \rho \rangle$. There is an edge in an RPG from $\langle X, l, \rho \rangle$ to $\langle X', l', \rho' \rangle$ if there exists a node $\langle X, l, \pi \rangle \in \langle X, l, \rho \rangle$ having a successor $\langle X', l', \pi' \rangle \in \langle X', l', \rho' \rangle$ (in terms of the infinite timed product graph). An RPG is called *stable* if whenever there is an edge $\langle X, l, \rho \rangle \longrightarrow \langle X', l', \rho' \rangle$, every node $\langle X, l, \pi \rangle \in \langle X, l, \rho \rangle$ has a successor in $\langle X', l', \rho' \rangle$. These definitions actually describe a family of RPGs corresponding to different choices of regions. Our local model checking algorithm will strive to construct the *smallest stable RPG* by making regions as coarse as possible.

We now argue that reasoning about RPGs during fixed point computation leads to an algorithmic (iterative) solution to the model checking problem. Let B be a block of an L_μ^t formula $\mathcal{B}$, with variables $\overline{X} = \{X_1, \ldots, X_n\}$ defined in B. Recall that we define the semantics of B in terms of the function $f_E^e(\overline{S})$, where $\overline{S}$ can be viewed as the restriction of environment e to $\overline{X}$ (see Section 3). We say that environment e is *composed of regions* if for every X, the set $\{\pi \mid \langle l, \pi \rangle \in e(X)\}$ is a finite union of regions.

Let m be the total number of clocks in a given instance of the model checking problem. The following lemma, a version of which also appears in [HNSY94], allows us to shift our attention from the dense lattice of tuples over $(\mathcal{L} \times \mathbf{R}^m)^n$, which was used to define the semantics of blocks in Section 3, to the discrete lattice over $(\mathcal{L} \times 2^{\Sigma})^n$.

Lemma 1. *If environment e is composed of regions, then $f_E^e(\overline{S})$ is also composed of regions.*

The proof of this lemma is similar to the one in [HNSY94]. By now noticing that the constants in the clock constraints that form the regions of $f_E^e(\overline{S})$ are no larger than the corresponding constants in e, we can limit our attention to those regions whose constants do not exceed the largest constant in the TSA and L_μ^t formula under consideration. Consequently, we need only consider finitely many regions, and the fixed points of the semantic function f_E^e become computable in an iterative manner [CC79].

It remains to be shown that given an RPG, the regions appearing in the RPG nodes are the right ones for computing the desired fixed points. This is captured by the following lemma, the proof of which is deferred to the full version of the paper. For an environment e composed of regions, we say that e *respects an RPG* $\mathcal{R}$ if for every X, there exists an $M \subseteq \{\langle l, \rho_i \rangle \mid \langle X, l, \rho_i \rangle \text{ is a node of } \mathcal{R}\}$ such that $e(X) = \bigcup M$.

Lemma 2. *Let $\mathcal{R}$ be a stable RPG. If e respects $\mathcal{R}$, then $f_E^e(\overline{S})$ also respects $\mathcal{R}$.*

The following observations further show how RPGs can be used to compute fixed points, and provide insight into our model checking algorithm. An assignment of boolean values to the nodes of an RPG is said to be a fixed point if the value of every $\wedge$-node (resp. $\vee$-node) is the conjunction (resp. disjunction) of the values of its successors. It is straightforward to check that an RPG fixed point is a fixed point for the corresponding L_μ^t formula. Moreover, if the RPG is acyclic, its fixed point is unique.

Consider, on the other hand, an RPG with cycles, and let C be one such cycle. Assume that all external successors of C (successors of nodes in C which themselves are not on C) have their values set in accordance with the desired fixed point. C is said to be free of *external interference* if the external successors of every $\vee$-node in C are false and, dually, the external successors of every $\wedge$-node in C are true. In this case, it is relatively easy to see that all nodes in C can be uniformly set to either true or false without violating the semantics of basic formulas: setting all nodes in the cycle to true will correspond to the greatest fixed point and setting all nodes in the cycle to false will correspond to the least fixed point.

5 The Local Model Checking Algorithm

This section presents *TMC*, our local model checking algorithm for L_μ^t. We begin with a general discussion of the algorithm followed by its pseudo-code. *TMC* is conceptually similar to existing untimed local model checking algorithms. The basic idea is that the value of the main variable of the formula in the start location depends on the values of the variables in adjacent locations, and the corresponding RPG nodes are constructed on-demand. The computation then unfolds in a similar fashion for newly constructed nodes, until sufficiently many nodes are constructed to determine the value of the RPG's start node.

On-the-Fly RPG Construction

TMC constructs the RPG on-the-fly: the RPG start node is constructed first and new successors are explored one node at a time. For efficiency purposes, the fixed-point computation is performed in tandem with the RPG construction by propagating the value of a newly explored node back to its predecessors. To ensure the correctness of the computation, the conditions imposed by Lemma 2 on the environment currently computed by the algorithm are satisfied at all times. For this purpose, *TMC* maintains the following two invariants:

I_e: whenever an edge $\langle X, l, \rho\rangle \longrightarrow \langle X', l', \rho'\rangle$ is present in the RPG, every node $\langle X, l, \pi\rangle \in \langle X, l, \rho\rangle$ has a successor in $\langle X', l', \rho'\rangle$.

I_v: the value of X in all timed product graph nodes in an RPG node $\langle X, l, \rho\rangle$ is the same. This value is recorded in a boolean variable associated with the RPG node, which is referred to as the *value* of the node.

I_e states that the constructed portion of the RPG is stable, while I_v asserts that the environment is of the right form. Although a step of the algorithm may temporarily invalidate the invariants, they are restored by splitting RPG nodes, which is discussed below.

TMC explores the RPG in a depth-first manner. A stack of nodes that have unvisited successors is kept. For each node appearing on the top of the stack, the algorithm iterates through its *untimed successors*. The pair $\langle X', l'\rangle$ is an untimed successor of RPG node $t = \langle X, l, \rho\rangle$ if t has a successor of the form $\langle X', l', \rho'\rangle$. It is assumed that the untimed successors of an RPG node are ordered in some way, so that the operation of choosing the next successor is meaningful. The only requirement we impose on the ordering is that if a node has successors whose variable components are defined in a block different from the block of the current node, all such successors precede any successor in the same block. If the next untimed successor of the current RPG node has not been previously visited, the largest possible region is taken to form the new RPG node; assuming the current node's location component is l, then this region is $L(l)$. Otherwise, the untimed node has been visited already and there is a set of RPG nodes corresponding to it. In this case, edges to appropriate RPG successors are introduced and the current node is split if needed to preserve I_e.

For every node the algorithm visits, an attempt is made to find a successor whose value will determine the value of the node, until all successors are visited. A predicate *done* is defined on RPG nodes, and $done(t)$ is set to true as soon as t's value is finalized. A node is allowed to derive its value only from *done* successors. When $done(t)$ becomes true of some node t, t's value is reported to its predecessors, which may lead to the finalization of their values as well.

Note that a node t belonging to an interference-free cycle does not have its value determined in this way: in this case, there would be a successor of t in the stack whose value depends on t. While exploring nodes, the algorithm identifies interference-free cycles by noticing that cycles cannot span block boundaries, and assigns to each node on such a cycle the value corresponding to the type of fixed point. Such assignments can be safely made after all successors of every node on the cycle are processed; i.e. no nodes belonging to the cycle remain in the stack of unexplored nodes.

Splitting RPG nodes

As RPG construction proceeds, more and more timing information, in the form of clock constraints contained in the TSA or L_μ^t formula, are encountered. Newly encountered clock constraints can invalidate the I_v and I_e invariants. To re-establish these invariants, certain RPG node regions must be partitioned into finer ones, thus effectively splitting the nodes. Since regions are defined as sets of constraints, it is easy to split a region ρ by a clock constraint $\mathbf{c}$: simply partition ρ into $\rho \cup \{\mathbf{c}\}$ and $\rho \cup \{\neg \mathbf{c}\}$. Note that one of the new regions may be empty, which means that no splitting has occurred. There are effective procedures, similar to those in [Dil89], to check for emptiness of a region and to "tighten" region boundaries by removing irrelevant constraints. To avoid redundant work, the algorithm never splits nodes for which *done* is true.

In the simplest case, the need for splitting arises when a new untimed node $\langle X, l \rangle$ is first encountered. Its default region $L(l)$ is split immediately in the following two cases:

- $X = A_t$. $L(l)$ is split in two subregions using the constraint A_t.
- $X([a]) \longrightarrow X'$ or $X(\langle a \rangle) \longrightarrow X'$. $L(l)$ is split into two or more subregions by applying the clock constraints on a-transitions leaving l.

Splitting $L(l)$ is accomplished in the algorithm by the procedure $\mathit{Split}(\langle X, l, L(l) \rangle)$. Note that the first case above violates I_v, while the second violates I_e.

A violation of I_e may arise every time a node $w = \langle X_w, l_w, \rho_w \rangle$ is split, necessitating the splitting of a predecessor $t = \langle X_t, l_t, \rho_t \rangle$ of w. The algorithms checks for such violations and carries out the required splittings by calling procedure $\mathit{Split}(t, w)$. Assume that X_t is not defined by a time successor operator ($\forall$ or $\exists$) and t has not been previously split.[3] Procedure $\mathit{Split}(t, w)$ computes the set of constraints to split t from those splitting w in the following way. First, the set ξ of clocks that were reset in moving from t to w is detected. This is the set of clocks labeling the edge $X_t \longrightarrow X_w$ in the formula graph and, if X_t is a transition modality ($\langle a \rangle$ or $[a]$), those labeling the corresponding transition in the TSA. Then all constraints splitting w are *projected* onto $\xi = 0$, i.e.

- constraints of the form $x \, R \, c, x \in \xi$ or $x + c \, R \, y + d, x, y \in \xi$ are discarded.
- constraints of the form $x + c \, R \, y + d, x \in \xi, y \notin \xi$ become $c - d \, R \, y$.
- all other constraints are left unchanged.

The resulting set of constraints is used to split t, and an edge $w' \longrightarrow t'$ is introduced as appropriate between subnodes w' of w and t' of t to satisfy I_e. For every new edge $t' \longrightarrow w'$, $\mathit{Split}(t', w')$ is recursively invoked.

The example in Figure 2 illustrates how *Split* works. We are given RPG nodes w and t with $t \longrightarrow w$; only their regions are depicted for the purpose of illustration. Assume that clock z is reset in going from t to w, and that w was split by the constraints $x \leq c$ and $z + d \leq x$ (the graph on the left). The projection of the constraints onto $z = 0$ leaves the first one intact and transforms the second into $d \leq x$. The dotted lines show how these constraints extend to t, and the resulting constraints are used to split t (the graph on the right).

[3] If t has been previously split by another successor, the splitting procedure must be applied to every subnode of t.

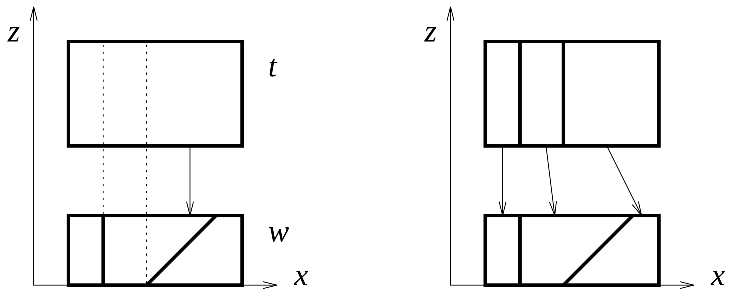

Fig. 2. Splitting an RPG node.

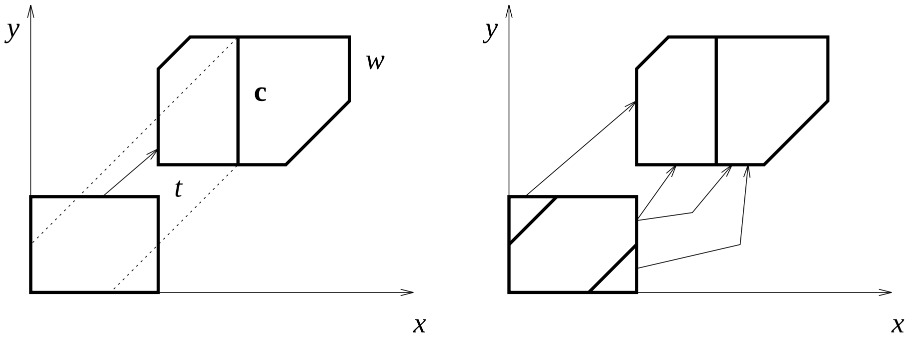

Fig. 3. Splitting an RPG node by a time successor.

When X_t is defined by a time successor operator, additional clock constraints must be taken into account. If w was split with the constraint $\mathbf{c} = x\ R\ d$, then for every constraint $y\ R'\ d'$ $(x \neq y)$ forming w, the new constraint $x - d\ R''\ y - d'$ (where R'' is derived from R and R' in the obvious way) is used to split t. For each clock y, there may be at most two constraints of this kind. This type of splitting is illustrated in Figure 3. Note that the number of successors may change for some parts of t. The case where the region of t is contained within the region of w is slightly special: $\mathbf{c}$ itself is used to split t, and only the subnode of t corresponding to $x < d$ is split with the new constraints described above.

The Pseudo-code

We now present the pseudo-code for *TMC*, our local model checking algorithm. Let $T = \langle \mathcal{L}, Act, \xi, \longrightarrow, l_0, L \rangle$ be a TSA and $\mathcal{B}$ an L^t_μ formula with X_0 as its main variable. Moreover, let t_0 denote an RPG node that contains $\langle X_0, l_0, \pi_0 \rangle$, the main variable of the formula in the start state of the timed system. Initially, $t_0 = \langle X_0, l_0, L(l_0) \rangle$. Note that t_0 can change (become smaller) as the algorithm proceeds.

We employ the following data structures: U is the depth-first stack of RPG nodes; P is the set of RPG nodes used to store nodes that may lie on interference-free cycles; D and S are sets of RPG nodes used in the propagation of values from successors to predecessors, and in the propagation of splits, respectively. All RPG nodes constructed by the algorithm are maintained in a search data structure, which also keeps track of the history of splits (a similar data structure is used in [YL93]).

```
procedure TMC(T, ℬ)
place t₀ on U
while U ≠ ∅
    t := first(U)
    if t has no successors then
        decide t's value
        done(t) := true
        move t from U to D
        processD
    else if ¬done(t) then
        if t has unexplored successors then
            choose the next untimed successor ⟨X, l⟩ of t
            if no nodes of the form ⟨X, l, ρ⟩ have been visited yet
                construct w = ⟨X, l, L(l)⟩
                add edge t⟶w
                place Split(w) on U
                processS
            else
                replace t with Split(t, ⟨X, l, L(l)⟩) in U
                processS
        else
            move t from U to P
            if U is empty or (X_t and X_first(U) are in different blocks) then
                while P ≠ ∅                          # process interference-free cycles
                    remove some t from P
                    if ¬done(t) then
                        done(t) := true
                        add t to D
                        processD
    else remove t from U
```

Procedure *processD*, which propagates the finalized values of nodes to their predecessors, is given by:

```
while D ≠ ∅
    remove some w from D
    decide w's value
    if w = t₀ then stop
    else for each predecessor t of w, and subnode t' of t
        report the value of w to t'
        if done(t') then
            add t' to D
```

The structure of procedure *processS*, not presented here, is similar to that of *processD*. It removes recently split nodes from the set S and propagates the effects of this splitting to their predecessors by calling *Split*.

Theorem 3. *Let T be a timed automaton and $\mathcal{B}$ an L_μ^t formula. When TMC terminates, the value of t_0 is true iff $\langle l_0, \pi_0 \rangle \in [\![X_0]\!][\![\mathcal{B}]\!]$.*

The proof consists of two independent parts. The first part shows that I_e and I_v are properly maintained by procedures *Split* and *processS*. The second part performs a case analysis on when the predicate *done* becomes true of an RPG node, and shows that the node assumes the correct value at that time. The proof itself will appear in the full version of the paper.

With regard to *TMC*'s efficiency, we have noted that splits of RPG nodes are performed only when necessary to accommodate a clock constraint appearing in T or $\mathcal{B}$. As we do not necessarily construct the complete RPG, the order in which the successors of a given node are visited may affect the size of the RPG we build on-the-fly.

6 Conclusions

We have presented a local model checking algorithm for a timed extension of the modal mu-calculus, the first local verification algorithm for real-time systems of which we are aware. Our algorithm, *TMC*, constructs the region product graph on the fly and makes regions as large as possible by performing a split operation only when dictated by a relevant clock constraint.

We are currently implementing *TMC* in the Concurrency Factory; because of the similarity with the search strategy employed in the untimed case, we are able to reuse a significant amount of code. The full version of the paper will report on the performance of *TMC* on some well-known benchmarks (cf. [ACD+92]).

Acknowledgements

We would like to thank the anonymous referees for their helpful comments.

References

[ACD+92] R. Alur, C. Courcoubetis, D. Dill, N. Halbwachs, and H. Wong-Toi. "An Implementation of Three Algorithms for Timing Verification Based on Automata Emptiness". In *Proceedings of IEEE Real-Time Symposium*. IEEE Computer Society Press, 1992.

[ACD93] R. Alur, C. Courcoubetis, and D. Dill. "Model-Checking for Real-Time Systems". *Information and Computation*, 104(1):2–34, 1993.

[ACH+92] R. Alur, C. Courcoubetis, N. Halbwachs, D. Dill, and H. Wong-Toi. "Minimization of Timed Transition Systems". In *Proceedings of CONCUR'92*. LNCS 630, 1992.

[AD94] R. Alur and D. L. Dill. "The Theory of Timed Automata". *Theoretical Comput. Sci.*, 126(2), 1994.

[Alu91] R. Alur. *Techniques for Authomatic Verification of Real-Time Systems.* PhD thesis, Stanford University, 1991.

[And94] H. R. Andersen. "Model Checking and Boolean Graphs". *Theoretical Comput. Sci.*, 126(1), 1994.

[BFH+92] A. Bouajjani, J.-C. Fernandez, N. Halbwachs, P. Raymond, and C. Ratel. "Minimal State Graph Generation". *Sci. Comput. Programming*, 18(3):247–269, 1992.

[BVW94] O. Bernholz, M. Y. Vardi, and P. Wolper. "An Automata-Thoeretic Approach to Branching-Time Model Checking". In *Proceedings of CAV'94*. LNCS 818, 1994.

[CC79] P. Cousot and R. Cousot. "Constructive Versions of Tarski's Fixed Point Theorems". *Pacific J. Math.*, 82(1):43–57, 1979.

[CE81] E. M. Clarke and E. A. Emerson. *Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic*. LNCS 131, 1981.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. "Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications". *ACM Trans. Prog. Lang. Syst.*, 8(2), 1986.

[CGL+94] R. Cleaveland, J. N. Gada, P. M. Lewis, S. A. Smolka, O. V. Sokolsky, and S. Zhang. "The Concurrency Factory – Practical Tools for Specification, Simulation, Verification and Implementation of Concurrent Systems". In *Proceedings of the DIMACS Workshop on Specification Techniques for Concurrent Systems, Princeton, NJ.*, 1994.

[Cle90] R. Cleaveland. "Tableau-Based Model Checking in the Propositional Mu-Calculus". *Acta Inf.*, 27, 1990.

[CPS93] R. Cleaveland, J. Parrow, and B. Steffen. "The Concurrency Workbench: A Semantics-Based Tool for the Verification of Concurrent Systems". *ACM TOPLAS*, 15(1), 1993.

[CS93] R. Cleaveland and B. Steffen. "A Linear-Time Model Checking Algorithm for the Alternation-Free Modal Mu-Calculus". *Formal Methods in System Design*, 2, 1993.

[Dil89] D. Dill. "Timing Assumptions and Verification of Finite-State Concurrent Systems". In *Proceedings of CAV'89*. LNCS 407, 1989.

[EJS93] E. A. Emerson, C. S. Jutla, and A. P. Sistla. "On Model Checking for Fragments of μ-calculus". In *Proceedings of CAV'93*. LNCS 697, 1993.

[EL86] E. A. Emerson and C.-L. Lei. "Efficient Model Checking in Fragments of the Propositional Mu-Calculus". In *Proceedings LICS '86*. IEEE Computer Society Press, 1986.

[Eme91] E. A. Emerson. "Real-Time and the Mu-Calculus". In *Real-Time: Theory in Practice*. LNCS 600, 1991.

[HLW91] U. Holmer, K. G. Larsen, and Y. Wang. "Deciding Properties of Regular Real Timed Processes". In *Proceedings of CAV'91*. LNCS 575, 1991.

[HNSY94] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. "Symbolic Model Checking for Real-time Systems". *Information and Computation*, 111(2), 1994.

[Koz83] D. Kozen. "Results on the Propositional Mu-Calculus". *Theoretical Comput. Sci.*, 27:333–354, 1983.

[Lar92] K. G. Larsen. "Efficient Local Correctness Checking". In *Proceedings of CAV'92*. LNCS 663, 1992.

[SS94] O. Sokolsky and S. A. Smolka. "Incremental Model Checking in the Modal Mu-Calculus". In *Proceedings of CAV'94*. LNCS 818, 1994.

[SW91] C. Stirling and D. Walker. "Local Model Checking in the Modal Mu-Calculus". *Theoretical Comput. Sci.*, 89(1), 1991.

[VLAP94] B. Vergauwen, J. Lewi, I. Avau, and A. Pote. "Efficient Computation of Nested Fix-Points, with applications to Model Checking". In *Proceedings of ICTL'94, 1st Intl. Conference on Temporal Logic*. LNCS 827, 1994.

[VW86] M. Y. Vardi and P. Wolper. "An Automata-Theoretic Approach to Automatic Program Verification". In *Proceedings of LICS'86*, pages 322–331. IEEE Computer Society Press, 1986.

[Wan91] Y. Wang. *A Calculus of Real Time Systems*. PhD thesis, Chalmers University of Technology, 1991.

[YL93] M. Yannakakis and D. Lee. "An Efficient Algorithm for Minimizing Real-Time Transition Systems". In *Proceedings of CAV'93*. LNCS 697, 1993.