Model Checking via Reachability Testing for Timed Automata

Luca Aceto[1] *, Augusto Burgueño[2] ** and Kim G. Larsen[1]

[1] BRICS***, Department of Computer Science, Aalborg University,
Fredrik Bajers Vej 7-E, DK-9220 Aalborg Ø, Denmark.
Email: {luca,kgl}@cs.auc.dk, Fax: +45 98 15 98 89
[2] ONERA-CERT, Département d'Informatique,
2 av. E. Belin, BP4025, 31055 Toulouse Cedex 4, France.
Email: a.burgueno@acm.org, Fax: +33 5 62 25 25 93

Abstract. In this paper we develop an approach to model-checking for timed automata via reachability testing. As our specification formalism, we consider a dense-time property language with clocks. This property language may be used to express safety and bounded liveness properties of real-time systems. We show how to automatically synthesize, for every formula φ, a *test automaton* T_φ in such a way that checking whether a system S satisfies the property φ can be reduced to a reachability question over the system obtained by making T_φ interact with S.

1 Introduction

Model-checking of real time systems has been extensively studied in the last few years, leading to both important theoretical results, setting the limits of decidability [3, 10], and to the emergence of practical tools as HyTech [9], Kronos [18] and UPPAAL [6], which have been successfully applied to the verification of real sized systems [5, 12].

The main motivation for the work presented in this paper stems from our experience with the verification tool UPPAAL. In such a tool, real-time systems are specified as networks of timed automata [3], which are then the object of the verification effort. The core of the computational engine of UPPAAL consists of a collection of efficient algorithms that can be used to perform reachability analysis over a model of an actual system. Any other kind of verification problem that the user wants to ask UPPAAL to perform must be encoded as a suitable reachability question. A typical example of such a problem is that of *model checking*. Experience has shown that it is often convenient to describe desired system properties as formulae of some real-time variant of standard modal or

* Partially supported by the Human Capital and Mobility project EXPRESS.
** Partially supported by Research Grant of the Spanish Ministry of Education and Culture and by BRICS. This work was carried out while the author was visiting Aalborg University.
*** Basic Research in Computer Science.

temporal logics (see, e.g., [4, 11, 15]). The model-checking problem then amounts to deciding whether a given system specification has the required property or not.

The way model-checking of properties other than plain reachability ones may currently be carried out in UPPAAL is as follows. Given a property φ to model-check, the user must provide a *test automaton* T_φ for that property. This test automaton must be such that the original system has the property expressed by φ if, and only if, none of the distinguished reject states of T_φ can be reached when the test automaton is made to interact with the system under investigation.

As witnessed by existing applications of this approach to verification by model-checking (cf., e.g., [13]), the construction of a test automaton from a temporal formula or informally specified requirements is a task that, in general, requires a high degree of ingenuity, and is error-prone. It would therefore be useful to automate this process by providing a compilation procedure from formulae in some sufficiently expressive real-time logic into appropriate test automata, and establishing its correctness once and for all. Apart from its practical and theoretical interest, the existence of such a connection between specification logics and automata would also free the average user of a verification tool like UPPAAL from the task of having to generate ad hoc test automata in his/her verifications based on the model-checking approach. We envisage that this will help make the tool usable by a larger community of designers of real-time systems.

In this paper we develop an approach to model-checking for timed automata via reachability testing. As our specification formalism, we consider a dense-time property language with clocks, which is a fragment of the one presented in [15]. This property language may be used to express safety and bounded liveness properties of real-time systems. We show how to automatically synthesize, for every formula φ, a so-called *test automaton* T_φ in such a way that checking whether a system S satisfies the property φ can be reduced to a reachability question over the system obtained by making T_φ interact with S. More precisely, we show that S satisfies property φ iff none of the distinguished reject nodes of the test automaton can be reached in the combined system $S \parallel T_\varphi$ (Thm. 5.2). This result is obtained for a model of timed automata with urgent actions and the interpretation of parallel composition used in UPPAAL.

The property language we consider in this paper only allows for a restricted use of the boolean 'or' operator, and of the diamond modality of Hennessy-Milner logic [8]. We argue that these restrictions are necessary to obtain testability of the property language, in the sense outlined above (Propn. 5.4). Indeed, as it will be shown in a companion paper [1], the property language presented in this study is remarkably close to being completely expressive with respect to reachability properties. In fact, a slight extension of the property language considered here allows us to reduce any reachability property of a composite system $S \parallel T$ to a model-checking problem of S.

Despite the aforementioned restrictions, the testable property language we consider is both of practical and theoretical interest. On the practical side, we have used the property language, and the associated approach to model-checking

via reachability testing it supports, in the specification and verification in UPPAAL of a collision avoidance protocol. This protocol was originally analyzed in [13], where rather complex test automata were derived in an ad hoc fashion from informal specifications of the expected behaviour of the protocol. The verification we present here is based on our procedure for the automatic generation of test automata from specifications. This has allowed us to turn specifications of the expected behaviour of the protocol into automata, whose precise fit with the original properties is guaranteed by construction. On the theoretical side, we have shown that the property language is powerful enough to permit the definition of *characteristic properties* [19], with respect to a timed version of the ready simulation preorder [16], for nodes of deterministic, τ-free timed automata. (This result is omitted from this extended abstract for lack of space, but see [2].)

This study establishes a connection between a property language for the specification of safety and bounded liveness properties of real-time systems and the formalism of timed automata. Our emphasis is on the reduction of the model-checking problem for the property language under consideration to an intrinsically automata-theoretic problem, viz. that of checking for the reachability of some distinguished nodes in a timed automaton. The blueprint of this endeavour lies in the automata-theoretic approach to the verification of finite-state reactive systems pioneered by Vardi and Wolper [20, 21]. In this approach to verification, the intimate relationship between linear time propositional temporal logic and ω-automata is exploited to yield elegant and efficient algorithms for the analysis of specifications, and for model-checking. The work presented in this paper is not based on a similarly deep mathematical connection between the property language and timed automata (indeed, it is not clear that such a connection exists because, as shown in [3], timed Büchi automata are not closed under complementation), but draws inspiration from that beautiful theory. In particular, the avenue of investigation pursued in this study may be traced back to the seminal [20].

The paper is organized as follows. We begin by introducing timed automata and timed labelled transition systems (Sect. 2). The notion of test automaton considered in this paper is introduced in Sect. 3, together with the interaction between timed automata and tests. We then proceed to present a real-time property language suitable for expressing safety and bounded liveness properties of real-time systems (Sect. 4). The step from properties to test automata is discussed in Sect. 5, and its implementation in UPPAAL in Sect. 6. Section 7 is devoted to a brief description of the specification and verification of a collision avoidance protocol using the theory developed in this paper. The paper concludes with a mention of some further results we have obtained on the topic of this paper, and a discussion of interesting subjects for future research (Sect. 8).

2 Preliminaries

We begin by briefly reviewing the timed automaton model proposed by Alur and Dill [3].

Timed Labelled Transition Systems Let $\mathcal{A}$ be a finite set of actions ranged over by a. We assume that $\mathcal{A}$ comes equipped with a mapping $\bar{\cdot} : \mathcal{A} \to \mathcal{A}$ such that $\bar{\bar{a}} = a$ for every $a \in \mathcal{A}$. We let $\mathcal{A}_\tau$ stand for $\mathcal{A} \cup \{\tau\}$, where τ is a symbol not occurring in $\mathcal{A}$, and use μ to range over it. Following Milner [17], τ will stand for an internal action of a system. Let $\mathbb{N}$ denote the set of natural numbers and $\mathbb{R}_{\geq 0}$ the set of non-negative real numbers. We use $\mathcal{D}$ to denote the set of delay actions $\{\epsilon(d) \mid d \in \mathbb{R}_{\geq 0}\}$, and $\mathcal{L}$ to stand for the union of $\mathcal{A}_\tau$ and $\mathcal{D}$.

Definition 2.1. A *timed labelled transition system* (TLTS) is a structure $\mathcal{T} = \langle S, \mathcal{L}, s^0, \longrightarrow \rangle$ where S is a set of *states*, $s^0 \in S$ is the initial state, and $\longrightarrow \subseteq S \times \mathcal{L} \times S$ is a transition relation satisfying the following properties:

- (TIME DETERMINISM) for every $s, s', s'' \in S$ and $d \in \mathbb{R}_{\geq 0}$, if $s \xrightarrow{\epsilon(d)} s'$ and $s \xrightarrow{\epsilon(d)} s''$, then $s' = s''$;
- (TIME ADDITIVITY) for every $s, s'' \in S$ and $d_1, d_2 \in \mathbb{R}_{\geq 0}$, $s \xrightarrow{\epsilon(d_1+d_2)} s''$ iff $s \xrightarrow{\epsilon(d_1)} s' \xrightarrow{\epsilon(d_2)} s''$, for some $s' \in S$;
- (0-DELAY) for every $s, s' \in S$, $s \xrightarrow{\epsilon(0)} s'$ iff $s = s'$.

Following [22], we now proceed to define versions of the transition relations that abstract away from the internal evolution of states as follows:

$$s \stackrel{a}{\Longrightarrow} s' \quad \text{iff} \quad \exists s''.\ s \xrightarrow{\tau}{}^{*}\ s'' \xrightarrow{a} s'$$

$$s \stackrel{\epsilon(d)}{\Longrightarrow} s' \quad \text{iff} \quad \text{there exists a computation}$$

$$s = s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} s_n = s' \quad (n \geq 0) \text{ where}$$

$$(a) \quad \forall i \in \{1, .., n\}.\ \alpha_i = \tau \text{ or } \alpha_i \in \mathcal{D}$$

$$(b) \quad d = \sum\{d_i \mid \alpha_i = \epsilon(d_i)\}$$

By convention, if the set $\{d_i \mid \alpha_i = \epsilon(d_i)\}$ is empty, then $\sum\{d_i \mid \alpha_i = \epsilon(d_i)\}$ is 0. With this convention, the relation $\stackrel{\epsilon(0)}{\Longrightarrow}$ coincides with $\xrightarrow{\tau}{}^{*}$, i.e., the reflexive, transitive closure of $\xrightarrow{\tau}$. Note that the derived transition relation $\stackrel{a}{\Longrightarrow}$ only abstracts from internal transitions *before* the actual execution of action a.

Definition 2.2. Let $\mathcal{T}_i = \langle \Sigma_i, \mathcal{L}, s_i^0, \longrightarrow_i \rangle$ $(i \in \{1, 2\})$ be two TLTSs. The parallel composition of $\mathcal{T}_1$ and $\mathcal{T}_2$ is the TLTS

$$\mathcal{T}_1 \parallel \mathcal{T}_2 = \langle \Sigma_1 \times \Sigma_2, \mathcal{D} \cup \{\tau\}, (s_1^0, s_2^0), \longrightarrow \rangle$$

where the transition relation $\longrightarrow$ is defined by the rules in Table 1. In Table 1, and in the remainder of the paper, we use the more suggestive notation $s \parallel s'$ in lieu of (s, s').

(1)	$\dfrac{s_1 \xrightarrow{\tau} s_1'}{s_1 \| s_2 \xrightarrow{\tau} s_1' \| s_2}$	(2) $\dfrac{s_2 \xrightarrow{\tau} s_2'}{s_1 \| s_2 \xrightarrow{\tau} s_1 \| s_2'}$
	(3) $\dfrac{s_1 \xrightarrow{a} s_1' \quad s_2 \xrightarrow{\overline{a}} s_2'}{s_1 \| s_2 \xrightarrow{\tau} s_1' \| s_2'}$	
(4)	$\dfrac{s_1 \xrightarrow{\epsilon(d)} s_1' \quad s_2 \xrightarrow{\epsilon(d)} s_2'}{s_1 \| s_2 \xrightarrow{\epsilon(d)} s_1' \| s_2'}$	$\forall t \in [0,d[, a \in \mathcal{A}, s_1'', s_2''.$ $\neg(s_1 \xrightarrow{\epsilon(t)} s_1'' \xrightarrow{a} \wedge\ s_2 \xrightarrow{\epsilon(t)} s_2'' \xrightarrow{\overline{a}})$

where s_i, s_i', s_i'' are states of $\mathcal{T}_i$ $(i \in \{1,2\})$,
$a, \overline{a} \in \mathcal{A}$ and $d, t \in \mathbb{R}_{\geq 0}$.

Table 1: Rules defining the transition relation $\rightarrow$ in $\mathcal{T}_1 \parallel \mathcal{T}_2$

This definition of parallel composition forces the composed TLTSs to synchronize on actions (all but τ-actions) and delays, but with the particularity that delaying is only possible when no synchronization on actions is. This amounts to requiring that all actions in $\mathcal{A}$ be *urgent*. The reader familiar with TCCS [22] may have noticed that the above definition of parallel composition precisely corresponds to a TCCS parallel composition in which all the actions in $\mathcal{A}$ are restricted upon. The use of this kind of parallel composition yields closed systems, of the type that can be analyzed using UPPAAL [6], and is inspired by the pioneering work by De Nicola and Hennessy on testing equivalence for processes [7].

Timed Automata Let C be a set of clocks. We use $\mathcal{B}(C)$ to denote the set of boolean expressions over atomic formulae of the form $x \sim p$, $x - y \sim p$, with $x, y \in C$, $p \in \mathbb{N}$, and $\sim \in \{<, >, =\}$. A *time assignment*, or *valuation*, v for C is a function from C to $\mathbb{R}_{\geq 0}$. For every time assignment v and $d \in \mathbb{R}_{\geq 0}$, we use $v + d$ to denote the time assignment which maps each clock $x \in C$ to the value $v(x) + d$. For every subset of clocks C', $[C' \rightarrow 0]v$ denotes the assignment for C which maps each clock in C' to the value 0 and agrees with v over $C \backslash C'$. Given a condition $g \in \mathcal{B}(C)$ and a time assignment v, the boolean value $g(v)$ describes whether g is satisfied by v or not.

Definition 2.3. A *timed automaton* is a tuple $A = \langle \mathcal{A}_\tau, N, n_0, C, E \rangle$ where N is a finite set of *nodes*, n_0 is the *initial node*, C is a finite set of *clocks*, and $E \subseteq N \times N \times \mathcal{A}_\tau \times 2^C \times \mathcal{B}(C)$ is a set of *edges*. The tuple $e = \langle n, n_e, \mu, r_e, g_e \rangle \in E$ stands for an edge from node n to node n_e (the *target* of e) with action μ, where r_e denotes the set of clocks to be reset to 0 and g_e is the enabling condition (or *guard*) over the clocks of A.

Example 2.4. The timed automaton depicted in Figure 1 has five nodes labelled n_0 to n_4, one clock x, and four edges. The edge from node n_1 to node n_2, for example, is guarded by $x \geq 0$, implies synchronization on a and resets clock x.

A *state* of a timed automaton A is a pair $\langle n, v \rangle$ where n is a node of A and v is a time assignment for C. The initial state of A is $\langle n_0, v_0 \rangle$ where n_0 is the initial node of A and v_0 is the time assignment mapping all clocks in C to 0.

The operational semantics of a timed automaton A is given by the TLTS $\mathcal{T}_A = \langle \Sigma, \mathcal{L}, \sigma^0, \longrightarrow \rangle$, where Σ is the set of states of A, σ^0 is the initial state of A, and $\longrightarrow$ is the transition relation defined as follows:

$$\langle n, v \rangle \xrightarrow{\mu} \langle n', v' \rangle \text{ iff } \exists r, g.\ \langle n, n', \mu, r, g \rangle \in E \wedge g(v) \wedge v' = [r \to 0]v$$

$$\langle n, v \rangle \xrightarrow{\epsilon(d)} \langle n', v' \rangle \text{ iff } n = n' \text{ and } v' = v + d$$

where $\mu \in \mathcal{A}_\tau$ and $\epsilon(d) \in \mathcal{D}$.

3 Testing Automata

In this section we take the first steps towards the definition of *model checking via testing* by defining *testing*. Informally, testing involves the parallel composition of the tested automaton with a *test automaton*. The testing process then consists in performing reachability analysis in the composed system. We say that the tested automaton fails the test if a special reject state of the test automaton is reachable in the parallel composition from their initial configurations, and passes otherwise. The formal definition of testing then involves the definition of what a test automaton is, how the parallel composition is performed and when the test has failed or succeeded. We now proceed to make these notions precise.

Definition 3.1. A *test automaton* is a tuple $T = \langle \mathcal{A}, N, N_T, n_0, C, E \rangle$ where $\mathcal{A}$, N, n_0, C, and E are as in Definition 2.3, and $N_T \subseteq N$ is the set of *reject nodes*.

Intuitively, a test automaton T interacts with a tested system, represented by a TLTS, by communicating with it. The dynamics of the interaction between the tester and the tested system is described by the parallel composition of the TLTS that is being tested and of $\mathcal{T}_T$. We now define failure and success of a test as follows.

Definition 3.2. Let $\mathcal{T}$ be a TLTS and T be a test automaton.

- We say that a node n of T is reachable from a state $s_1 \parallel s_2$ of $\mathcal{T} \parallel \mathcal{T}_T$ iff there is a sequence of transitions leading from $s_1 \parallel s_2$ to a state whose $\mathcal{T}_T$ component is of the form $\langle n, u \rangle$.
- We say that a state s of $\mathcal{T}$ *fails the T-test* iff a reject node of T is reachable in $\mathcal{T} \parallel \mathcal{T}_T$ from the state $s \parallel \langle n_0, u_0 \rangle$, where $\langle n_0, u_0 \rangle$ is the initial state of $\mathcal{T}_T$. Otherwise, we say that s *passes the T-test*.

In the remainder of the paper, we shall mostly apply test automata to the TLTSs that give operational semantics to timed automata. In that case, we shall use the suggestive notation $A \parallel T$ in lieu of $\mathcal{T}_A \parallel \mathcal{T}_T$.

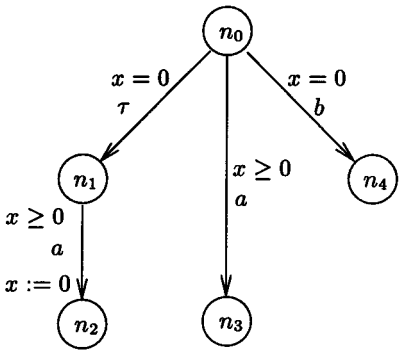


Figure 1: Timed automaton A

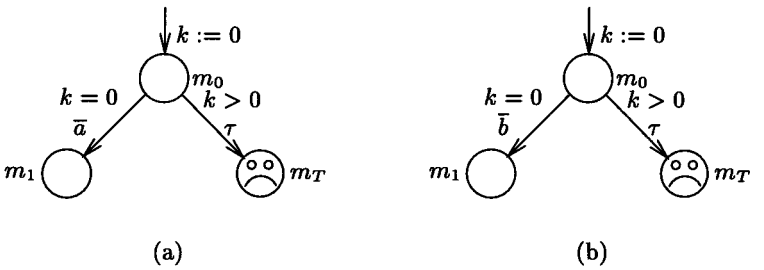


Figure 2: The test automata T_a and T_b

Example 3.3. Consider the timed automaton A of Figure 1 and the test automaton T_b of Figure 2(b). The reject node m_T of the test automaton is reachable from the initial state of $A \parallel T_b$, as follows:

1. first the automaton A can execute the τ-transition and go to node n_1, thus preempting the possibility of synchronizing on channel b with T,
2. now both automata can let time pass, thus enabling the τ-transition from node m_0 in T_b and making m_T reachable.

In this case we say that A fails the test. If we test A using the automaton T_a of Figure 2(a), then in all cases A and T_a must synchronize on a and no initial delay is possible. It follows that the reject node m_T of T_a is unreachable, and A passes the test.

4 The Property Language

We consider a dense-time property language with clocks, which is a fragment of the one presented in [15] and is suitable for the specification of safety and bounded liveness properties of TLTSs.

Definition 4.1. Let K be a set of clocks, disjoint from C. The set SBLL of (safety and bounded liveness) formulae over K is generated by the following grammar:

$$\begin{array}{rcl} \varphi & ::= & \mathtt{tt} \mid \mathtt{ff} \mid c \mid \varphi_1 \wedge \varphi_2 \mid c \vee \varphi \mid \forall\varphi \mid \\ & & [a]\varphi \mid \langle a\rangle\mathtt{tt} \mid x \,\underline{\mathtt{in}}\, \varphi \mid X \mid \mathtt{max}(X,\varphi) \\ c & ::= & x \sim p \mid x - y \sim p \end{array}$$

where $a \in \mathcal{A}$, $x, y \in K$, $p \in \mathbb{N}$, $\sim \in \{<, >, =\}$, X is a formula variable and $\mathtt{max}(X,\varphi)$ stands for the maximal solution of the recursion equation $X = \varphi$.

A *closed recursive formula* of SBLL is a formula in which every formula variable X appears within the scope of some $\mathtt{max}(X,\varphi)$ construct. In the remainder of this paper, every formula will be closed, unless specified otherwise.

Given a TLTS $\mathcal{T} = \langle S, \mathcal{L}, s^0, \longrightarrow\rangle$, we interpret the closed formulae in SBLL over extended states. An *extended state* is a pair $\langle s, u\rangle$ where s is a state of $\mathcal{T}$ and u is a time assignment for the formula clocks in K.

Definition 4.2. Consider a TLTS $\mathcal{T} = \langle S, \mathcal{L}, s^0, \longrightarrow\rangle$. The satisfaction relation $\models_w$ is the largest relation satisfying the implications in Table 2.

We say that $\mathcal{T}$ weakly satisfies φ, written $\mathcal{T} \models_w \varphi$, when $\langle s^0, u_0\rangle \models_w \varphi$, where u_0 is the time assignment mapping every clock in K to 0. In the sequel, for a timed automaton A, we shall write $A \models_w \varphi$ in lieu of $\mathcal{T}_A \models_w \varphi$.

The weak satisfaction relation is closed with respect to the relation $\xrightarrow{\tau}{}^*$, in the sense of the following proposition.

Proposition 4.3. *Let $\mathcal{T} = \langle S, \mathcal{L}, s^0, \longrightarrow\rangle$ be a TLTS. Then, for every $s \in S$, $\varphi \in$ SBLL and valuation u for the clocks in K, $\langle s, u\rangle \models_w \varphi$ iff, for every s' such that $s \xrightarrow{\tau}{}^* s'$, $\langle s', u\rangle \models_w \varphi$.*

The reader familiar with the literature on variations on Hennessy-Milner logic [17] and on its real-time extensions [23] may have noticed that our definition of the satisfaction relation is rather different from the standard one presented in the literature. For instance, one might expect the clause of the definition of the satisfaction relation for the formula $\langle a\rangle\mathtt{tt}$ to read

$$\langle s, u\rangle \models_w \langle a\rangle\mathtt{tt} \quad \text{implies} \quad s \stackrel{a}{\Longrightarrow} s' \text{ for some } s' \ . \tag{1}$$

Recall, however, that our main aim in this paper is to develop a specification language for timed automata for which the model checking problem can be effectively reduced to deciding reachability. More precisely, for every formula $\varphi \in$ SBLL, we aim at constructing a test automaton T_φ such that every extended state $\langle s, u\rangle$ of a timed automaton satisfies φ iff it passes the test T_φ (in a sense to be made precise in Defn. 5.1). With this aim in mind, a reasonable proposal for a test automaton for the formula $\langle a\rangle\mathtt{tt}$, interpreted as in (1), is the automaton

$$
\begin{array}{rcl}
\langle s,u\rangle \models_w \mathtt{tt} & \Rightarrow & \mathit{true} \\
\langle s,u\rangle \models_w \mathtt{ff} & \Rightarrow & \mathit{false} \\
\langle s,u\rangle \models_w c & \Rightarrow & c(u) \\
\langle s,u\rangle \models_w \varphi_1 \wedge \varphi_2 & \Rightarrow & \forall s'.\ s \xrightarrow{\tau}{}^{*} s' \text{ implies } \langle s',u\rangle \models_w \varphi_1 \text{ and } \langle s',u\rangle \models_w \varphi_2 \\
\langle s,u\rangle \models_w c \vee \varphi & \Rightarrow & \forall s'.\ s \xrightarrow{\tau}{}^{*} s' \text{ implies } c(u) \text{ or } \langle s',u\rangle \models_w \varphi \\
\langle s,u\rangle \models_w [a]\varphi & \Rightarrow & \forall s'.\ s \stackrel{a}{\Longrightarrow} s' \text{ implies } \langle s',u\rangle \models_w \varphi \\
\langle s,u\rangle \models_w \langle a\rangle \mathtt{tt} & \Rightarrow & \forall s'.\ s \xrightarrow{\tau}{}^{*} s' \text{ implies } s' \xrightarrow{a} s'' \text{ for some } s'' \\
\langle s,u\rangle \models_w \mathbf{V}\varphi & \Rightarrow & \forall d \in \mathbb{R}_{\geq 0}\ \forall s'.\ \ s \stackrel{\epsilon(d)}{\Longrightarrow} s' \ \text{ implies } \langle s',u+d\rangle \models_w \varphi \\
\langle s,u\rangle \models_w x \ \underline{\mathtt{in}}\ \varphi & \Rightarrow & \forall s'.\ \ s \xrightarrow{\tau}{}^{*} s' \text{ implies } \langle s',[\{x\}\to 0]u\rangle \models_w \varphi \\
\langle s,u\rangle \models_w \mathtt{max}(X,\varphi) & \Rightarrow & \forall s'.\ \ s \xrightarrow{\tau}{}^{*} s' \text{ implies } \langle s',u\rangle \models_w \varphi\{\mathtt{max}(X,\varphi)/X\}
\end{array}
$$

Table 2: Weak satisfaction implications

depicted in Figure 2(a). However, it is not hard to see that such an automaton could be brought into its reject node m_T by one of its possible interactions with the timed automaton associated with the TCCS agent $a + \tau$. This is due to the fact that, because of the definition of parallel composition we have chosen, a test automaton cannot prevent the tested state from performing its internal transition leading to a state where an a-action is no longer possible. (In fact, it is not too hard to generalize these ideas to show that *no* test automaton for the formula $\langle a\rangle\mathtt{tt}$ exists under the interpretation given in (1).) Similar arguments may be applied to all the formulae in the property language SBLL that involve occurrences of the modal operator $[a]$ and/or of the primitive proposition $\langle a\rangle\mathtt{tt}$.

The reader might have also noticed that the language SBLL only allows for a restricted use of the logical connective 'or'. This is due to the fact that it is impossible to generate test automata even for simple formulae like $\langle a\rangle\mathtt{tt} \vee [b]\mathtt{ff}$— cf. Propn. 5.4.

Notation. Given a state $\langle n, v\rangle$ of a timed automaton, and a valuation u for the formula clocks in K, we write $\langle n, v : u\rangle$ for the resulting extended state.

5 Model checking via testing

In Sect. 3 we have seen how we can perform tests on timed automata. We now aim at using test automata to determine whether a given timed automaton weakly satisfies a formula in L. As already mentioned, this approach to model checking for timed automata is not merely a theoretical curiosity, but it is the way in which model checking of properties other than plain reachability ones is routinely carried out in a verification tool like UPPAAL. In order to achieve our goal, we shall define a "compilation" procedure to obtain a test automaton from the formula we want to test for. By means of this compilation procedure, we

automate the process of generating test automata from specifications—a task which has so far required a high degree of ingenuity and is error-prone.

Definition 5.1. Let φ be a formula in SBLL and T_φ be a test automaton over clocks $\{k\} \cup K$, k fresh.

- For every extended state $\langle n, v : u\rangle$ of a timed automaton A, we say that $\langle n, v : u\rangle$ passes the T_φ-test iff no reject node of T_φ is reachable from the state $\langle n, v\rangle \parallel \langle m_0, \{k\} \to 0 : u\rangle$, where m_0 is the initial node of T_φ.
- We say that the test automaton T_φ *weakly tests* for the formula φ iff the following holds: for every timed automaton A and every extended state $\langle n, v : u\rangle$ of A, $\langle n, v : u\rangle \models_w \varphi$ iff $\langle n, v : u\rangle$ passes the T_φ-test.

Theorem 5.2. *For every closed formula φ in* SBLL, *there exists a test automaton T_φ that weakly tests for it.*

Proof. (SKETCH.) The test automata are constructed by structural induction on open formulae. (The UPPAAL implementation of the constructions is depicted in Figures 3 and 4.) It can be shown that, for every closed formula φ, the resulting automaton T_φ weakly tests for φ. The details of the proof will be presented in the full version of the paper.

Corollary 5.3. *Let A be a timed automaton. Then, for every $\varphi \in$ SBLL, there exists a test automaton T_φ with a reject node m_T such that $A \models_w \varphi$ iff node m_T is not reachable in $A \parallel T_\varphi$.*

As remarked in Sect. 4, the property language SBLL only allows for a restricted use of the 'or' operator. This is justified by the following negative result.

Proposition 5.4. *The formula $\langle a\rangle \mathtt{tt} \vee [b]\mathtt{ff}$ is not weakly testable.*

6 Implementation in UPPAAL

The UPPAAL *constructs* The implementation of testing using the parallel composition operator presented in Sect. 3 requires a model of communicating timed automata with *urgent actions* (cf. rule (4) in Table 1). This feature is available in the UPPAAL model. The test automata are inductively obtained from the formula in a constructive manner, according to the constructions shown in Figures 3 and 4. In these constructions all actions in $\mathcal{A}$ are intended to be urgent. As in UPPAAL it is not possible to guard edges labelled with urgent actions, the theoretical construction for $T_{[a]\varphi}$ used in the proof of Thm. 5.2 is implemented by means of node invariants.

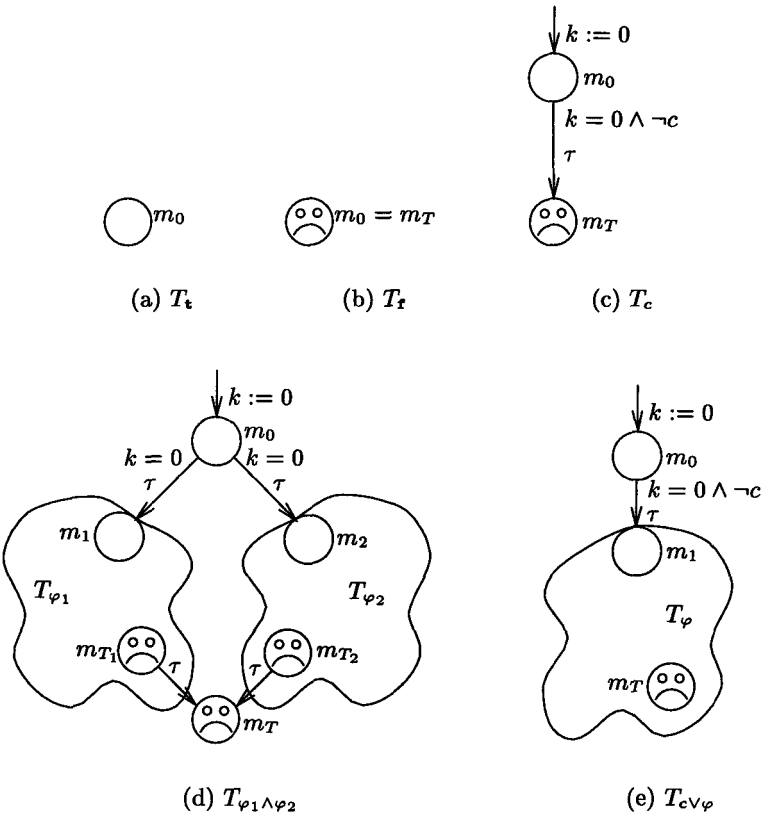
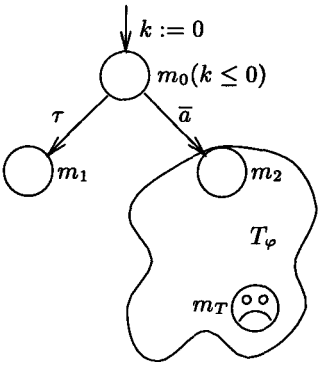
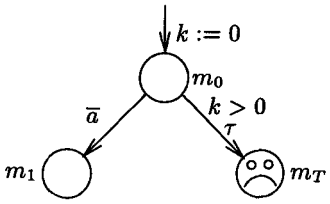


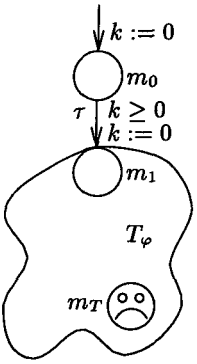
Figure 3: Test automata for SBLL sub-formulae



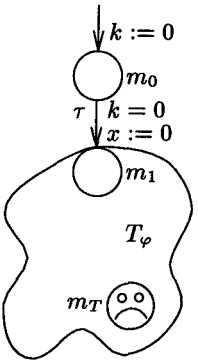
(a) $T_{[a]\varphi}$



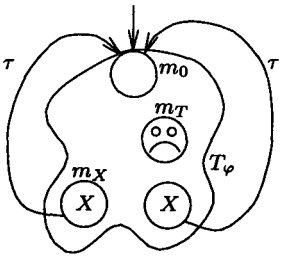
(b) $T_{\langle a \rangle \mathtt{t}}$



(c) $T_{\forall \varphi}$

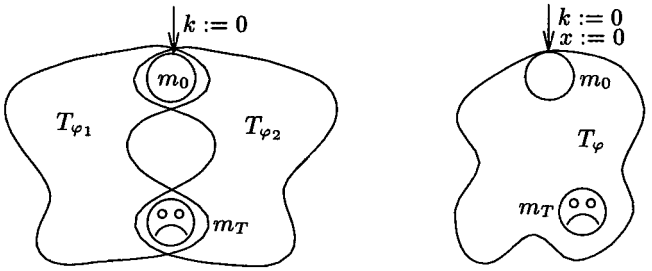


(d) $T_{x \text{ in } \varphi}$



(e) $T_{\max(X,\varphi)}$

Figure 4: Test automata for SBLL sub-formulae (cont.)



(a) $T_{\varphi_1 \wedge \varphi_2}$ (b) $T_{x \text{ in } \varphi}$

Figure 5: New simplified constructs

Simplification of the test automaton In certain cases, it is possible to optimize the construction of a test automaton from a formula by applying heuristics. Here we just remark on two possible simplifications. One is with respect to $T_{\varphi_1 \wedge \varphi_2}$ (Figure 3(d)) and the other one with respect to $T_{x \text{ in } \varphi}$ (Figure 4(d)). Both simplifications involve the elimination of the τ-transitions emanating from node m_0. This leads to the constructs shown in Figures 5(a) and 5(b). The test automaton of Figure 5(a) is obtained by setting the initial nodes of T_{φ_1} and T_{φ_2} to be the same node m_0, and the same for the reject node m_T. For $T_{x \text{ in } \varphi}$, the reset $x := 0$ is added to the incoming edge of T_φ. Nevertheless, these simplifications cannot be applied in general. For example, if the *and* operator involves the conjunction of $[a]\varphi$ and $\langle a \rangle \mathtt{tt}$, or $[a]\varphi$ and $\Box\varphi$, or $\langle a \rangle \mathtt{tt}$ and $\Box\varphi$, then the proposed simplification leads to incorrect test automata. This is because there is a different interpretation of evolving time in each operand, by, for example, leading to a reject state in one operand and to a safe one in the other one, or simply not being allowed in one case and being necessary in the other. Similarly, the in operator can be simplified only when it is not an operand in an *and* operation which has already been simplified.

High level operators The basic constructs of the logic SBLL can be used to define high level temporal operators, which may be used to simplify the writing of logical specifications (and substantiate our claim that SBLL can indeed express safety and bounded liveness properties). Here we confine ourselves to showing how to define the temporal operators `until`, `before` and `inv`:

$$\varphi \texttt{ until } c \stackrel{\text{def}}{=} \max(X, c \vee (\varphi \wedge \bigwedge_a [a]X \wedge \Box X))$$

$$\varphi \texttt{ until}_{\leq t} c \stackrel{\text{def}}{=} x \text{ in } ((\varphi \wedge x \leq t) \texttt{ until } c)$$

$$\texttt{before}_t\ c \stackrel{\text{def}}{=} \mathtt{tt} \texttt{ until}_{\leq t} c$$

$$\texttt{inv } \varphi \stackrel{\text{def}}{=} \max(X, \varphi \wedge \bigwedge_a [a]X \wedge \Box X)\ .$$

7 Example

Consider a number of stations connected on an Ethernet-like medium, following a basic CSMA/CD protocol as the one considered in [13]. On top of this basic protocol, we want to design a protocol without collisions (applicable for example to real time plants). In particular, we want to guarantee an upper bound on the transmission delay of a buffer, assuming that the medium does not lose or corrupt data, and that the stations function properly. The simplest solution is to introduce a dedicated master station which asks the other stations whether they want to transmit data to another station (see Figure 7). Such a master station has to take into account the possible buffer delays within the receiving stations to ensure that the protocol enjoys the following properties: (1) collision cannot occur, (2) the transmitted data eventually reach their destination, (3) data which are received have been transmitted by a sender, and (4) there is a known upper bound on the transmission delay, assuming error-free transmission.

Modelling and verification of such a protocol in UPPAAL has been presented in [13], where the details of such a modelling may be found. Here we only focus on the external view of the behaviour of the system. The observable actions are: user i sending a message, written `send_i!`, and user j receiving a message, written `recv_j!`, for $i, j = \{1, 2, 3\}$. The verification of the protocol presented in *op. cit.* was based on the ad hoc generation of test automata from informal specifications of system requirements. Indeed, some of the test automata that resulted from the informal requirements were rather complex, and it was difficult to extract their semantic interpretation. We now have at our disposal a precise property language to *formally* describe the expected behaviour of the protocol, together with an automatic compilation of such specifications into test automata, and we can therefore apply the aforementioned theory to test the behaviour of the protocol.

One of the requirements of the protocol is that there must be an upper bound on the transmission delay. Assuming that this upper bound is 4, this property can be expressed by means of the following formula in SBLL:

$$\texttt{inv}\left([\texttt{send\_1!}]s \ \underline{\texttt{in}} \ \mathbb{W}([\texttt{recv\_2!}](s < 4) \wedge [\texttt{recv\_3!}](s < 4))\right)$$

This formula states that it invariantly holds that whenever user 1 sends a message, it will be received by users 2 and 3 within 4 units of time. Note that we consider transmission to be error-free, so the message will eventually be received. What we are interested in is the delay expressed by clock s. The test automaton corresponding to this formula is shown in Figure 6. (Note that, although the formula above expresses the required behaviour of the protocol in a very direct way, its encoding as a test automaton is already a rather complex object—which we were glad not to have to build by hand!)

In order to experiment with our current implementation of the test automata construction in UPPAAL, we have also carried out the verification of several other properties of the protocol. For instance, we have verified that, under the assumption that the master waits for two time units before sending out its

enquiries, the protocol has a round-trip time bound of 18 time units, and that no faster round-trip exists. However, we have verified that changing the waiting time in the master to zero will allow for faster round-trip times. The details of these experiments will be reported in the full version of this study.

8 Concluding Remarks

As argued in, e.g., [24], efficient algorithms for deciding reachability questions can be used to tackle many common problems related to verification. In this study, following the lead of [20], we have shown how to reduce model-checking of safety and bounded liveness properties expressible in the real-time property language SBLL to checking for reachability of reject states in suitably constructed test automata. This approach allows us to take full advantage of the core of the computational engine of the tool UPPAAL [6], which consists of a collection of efficient algorithms that can be used to perform reachability analysis over timed automata.

The practical applicability of the approach to model-checking that we have developed in this paper has been tested on a basic CSMA/CD protocol. More experimental activity will be needed to fully test the feasibility of model-checking via reachability testing. So far, all the case studies carried out with the use of UPPAAL (see, e.g., [5, 13, 14]) seem to support the conclusion that this approach to model-checking can indeed be applied to realistic case studies, but further evidence needs to be accumulated to substantiate this claim. In this process of experimentation, we also expect to further develop a collection of heuristics that can be used to reduce the size of the test automata obtained by means of our automatic translation of formulae into automata.

In this study, we have shown how to translate the formulae in the property language SBLL into test automata in such a way that model-checking can be reduced to testing for reachability of distinguished reject nodes in the generated automata. Indeed the property language presented in this study is remarkably close to being completely expressive with respect to reachability properties. In fact, as it will be shown in a companion paper [1], a slight extension of the property language considered here allows us to reduce any reachability property for a composite system $S \parallel T$ to a model-checking problem of S.

The interpretation of the formulae in our specification formalism presented in Table 2 abstracts from the internal evolution of real-time processes in a novel way. A natural question to ask is whether the formulae in the property language SBLL are testable, in the sense of this paper, when interpreted with respect to the transition relation $\longrightarrow$. In the full version of this work, we shall show that this is indeed possible if the test automata are allowed to have *committed locations* [5], and the definition of the parallel composition operator is modified to take the nature of these locations into account. We expect, however, that the weak interpretation of the property language will be more useful in practical applications of our approach to model-checking.

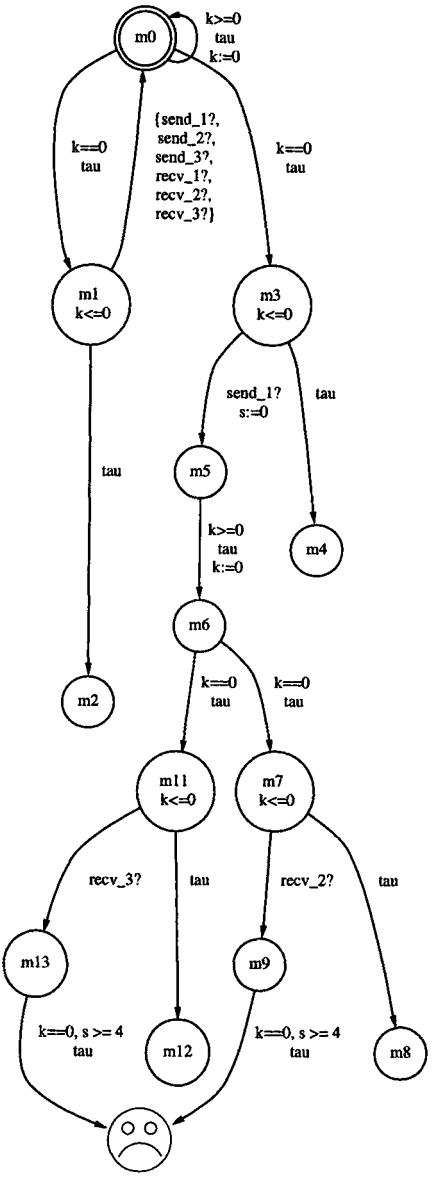


Figure 6: Test automaton for the invariant property

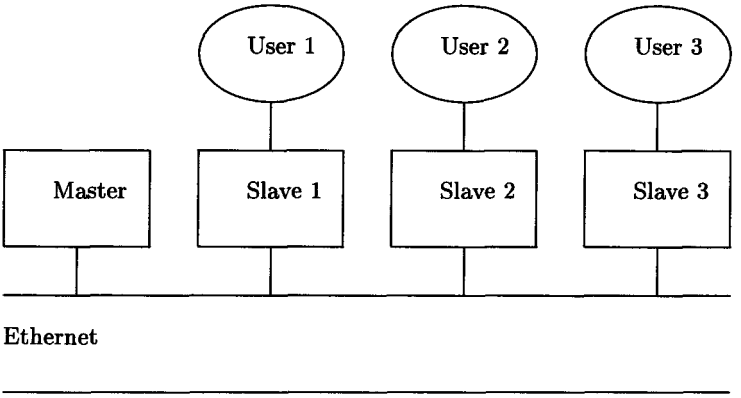


Figure 7: The Ethernet

Acknowledgements. We thank Patricia Bouyer for her help in the implementation of the tool, Kåre Jelling Kristoffersen for his proof-reading, and the anonymous referees for their comments.

References

1. L. Aceto, P. Bouyer, A. Burgueño, and K. G. Larsen. The limitations of testing for timed automata, 1997. Forthcoming paper.
2. L. Aceto, A. Burgueño, and K. G. Larsen. Model checking via reachability testing for timed automata. Research Report RS–97–29, BRICS, Aalborg University, November 1997.
3. R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.
4. R. Alur and T.A. Henzinger. A really temporal logic. *Journal of the ACM*, 41(1):181–204, 1994. Preliminary version appears in Proc. 30th FOCS, 1989.
5. J. Bengtsson, D. Griffioen, K. Kristoffersen, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Verification of an audio protocol with bus collision using UPPAAL. In R. Alur and T. A. Henzinger, editors, *Proc. of the 8th. International Conference on Computer-Aided Verification, CAV'96*, volume 1102 of *Lecture Notes in Computer Science*, New Brunswick, New Jersey, USA, July 31 – August 3 1996. Springer-Verlag.
6. J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL - a tool suite for automatic verification of real-time systems. In *Proc. of the 4th DIMACS Workshop on Verification and Control of Hybrid Systems*, New Brunswick, New Jersey, 22–24 October 1995.
7. R. De Nicola and M. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.
8. M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the Association for Computer Machinery*, 32(1):137–161, January 1985.
9. T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: the next generation. In *Proc. of the 16th Real-time Systems Symposium, RTSS'95*. IEEE Computer Society press, 1995.

10. T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata? In *Proc. of the 27th Annual ACM Symposium on Theory of Computing, STOC'95*, pages 373–382, 1995. Also appeared as Cornell University technical report TR95-1541.
11. T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.
12. P.-H. Ho and H. Wong-Toi. Automated analysis of an audio control protocol. In P. Wolper, editor, *Proc. of the 7th. International Conference on Computer-Aided Verification, CAV'95*, volume 939 of *Lecture Notes in Computer Science*, pages 381–394, Lige, Belgium, July 1995. Springer-Verlag.
13. H. E. Jensen, K. G. Larsen, and A. Skou. Modelling and analysis of a collision avoidance protocol using SPIN and UPPAAL. In *DIMACS Workshop SPIN '96, 2nd International SPIN Verification Workshop on Algorithms, Applications, Tool Use, Theory.* Rutgers University, New Jersey, USA, 1996.
14. K.J. Kristoffersen and P. Pettersson. Modelling and analysis of a steam generator using UPPAAL. In *Proc. of the 7th Nordic Workshop on Programming Theory*, Göteborg, Sweden, November 1–3 1995.
15. F. Laroussinie, K. G. Larsen, and C. Weise. From timed automata to logic - and back. In J. Wiedermann and P. Hájek, editors, *Proc. of the 20th. International Symposium on Mathematical Foundations of Computer Science, MFCS'95*, volume 969 of *Lecture Notes in Computer Science*, pages 529–539, Prague, Czech Republic, August 28 - September 1 1995. Springer-Verlag.
16. Kim G. Larsen and Arne Skou. Bisimulation through probabilistic testing. *Information and Computation*, 94(1):1–28, September 1991.
17. R. Milner. *Communication and Concurrency.* Series in Computer Science. Prentice Hall International, 1989.
18. A. Olivero and S. Yovine. *Kronos: a tool for verifying real-time systems. User's guide and reference manual.* VERIMAG, Grenoble, France, 1993.
19. B. Steffen and A. Ingólfsdóttir. Characteristic formulae for processes with divergence. *Information and Computation*, 110(1):149–163, April 1994.
20. M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. of the 1st. Annual Symposium on Logic in Computer Science, LICS'86*, pages 322–331. IEEE Computer Society Press, 1986.
21. M. Y. Vardi and P. Wolper. Reasoning about infinte computations. *Information and Computation*, 115:1–37, 1994.
22. Y. Wang. Real-time behaviour of asynchronous agents. In J.C.M. Baeten and J.W. Klop, editors, *Proc. of the Conference on Theories of Concurrency: Unification and Extension, CONCUR'90*, volume 458 of *Lecture Notes in Computer Science*, pages 502–520, Amsterdam, The Netherlands, August 27–30 1990. Springer-Verlag.
23. Y. Wang. *A calculus of real time systems.* PhD thesis, Chalmers university of Technology, Göteborg, Sweden, 1991.
24. P. Wolper. Where could SPIN go next? a unifying approach to exploring infinite state spaces. Slides for an invited talk at the *1997 SPIN Workshop*, Enschede, The Netherlands. Available at the URL `http://www.montefiore.ulg.ac.be/~pw/papers/psfiles/SPIN4-97.ps`.