Modeling for symbolic analysis of safety instrumented systems with clocks

Roland Kindermann, Tommi Junttila, and Ilkka Niemelä
Department of Information and Computer Science
Aalto University
PO Box 15400, FI-00076 Aalto, Finland
Email: Roland.Kindermann@tkk.fi, Tommi.Junttila@tkk.fi, Ilkka.Niemela@aalto.fi

***Abstract*—Safety instrumented systems (*SIS*) monitor industrial processes and automatically react on dangerous situations. SIS often consist of both logical and time-dependent building blocks. This paper introduces symbolic timed transition systems, a formalism designed for concise and modular description of SIS with clocks and similar time-dependent systems. Furthermore, an implementation of symbolic timed transition systems as an extension to NuSMV is devised. Two ways of checking properties on symbolic timed transition systems are developed: complete, region-abstraction-based model checking using binary decision diagrams and SMT-based bounded model checking. Both approaches are evaluated experimentally.**

***Keywords*-safety instrumented systems; symbolic model checking; NuSMV; region abstraction; timed systems**

I. Introduction

Digital Instrumentation and Control (I&C) systems are widely used in industry. In addition to a basic process control part, such an I&C system typically includes a Safety Instrumented System (SIS) that implements safety related functions, for example, achieving a safe state of the process when dangerous process conditions are detected. Because of its crucial role in guaranteeing safety, a SIS requires careful validation. However, it is very challenging to reach adequate coverage using traditional validation techniques based on testing because of a number of reasons: a typical SIS has to function reactively handling a large number of inputs changing independently, most safety functions are time-dependent, and when multiple different alarm conditions are occurring simultaneously, different safety functions reacting to the conditions are typically coordinated by timing.

Model checking [1] provides an alternative that is able to cover the behaviors of a SIS design more systematically than testing. We have studied the use of standard model checking tools such as NuSMV [2] and Uppaal [3] in verifying SIS designs given as functional block diagrams [4], [5]. Although the results have been promising, the tools leave room for improvement. For example, NuSMV enables a compact and modular way of modeling systems described using function block diagrams but does not support real time as used in typical SIS designs. On the other hand, Uppaal supports directly real time modeling. However, in our experience it scales poorly when the system has a large number of non-deterministic input variables and it does not lend itself well to modular and compact modeling of functional block diagrams.

In this paper we aim to develop a modeling formalism that combines the advantages of NuSMV and Uppaal type modeling languages but for which verification tools can be built exploiting known model checking methods and their advanced implementation techniques.

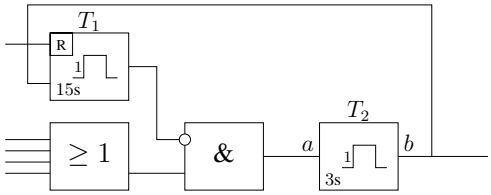
We propose a formalism called symbolic timed transition systems (STTS). The main difference of STTS, when compared to timed automata, is that it allows non-deterministic input variables and is not bound to automata structure and typical synchronization schemes. This enables a modular way of modeling timed functional block designs. We show that a timed automaton can be compactly expressed as an STTS. In order to apply model checking to STTS, we propose encodings that map an STTS into untimed symbolic transition systems by employing the region abstraction [6]. Such untimed symbolic transition systems can be model checked with tools such as NuSMV or VIS, either by applying BDD-based methods or with bounded model checking. We evaluate these encodings against each other as well as against bounded model checking with SMT solvers.

II. Symbolic Timed Transition Systems

Symbolic Timed Transition Systems are a formalism designed with convenient modeling of safety instrumented systems (SIS) in mind. SIS typically consist both of logical building blocks, like conjunctions or flip-flops, and of time-related building blocks like delays or pulses (see the left side of Fig. 1 for an example of a part of an SIS). For convenient modeling of such systems, STTS combine logic-based modeling, as is used by many (non-timed) symbolic model checkers, with the support for clocks similar to those used, e.g., in timed automata.

In the following, we use standard concepts of propositional and first-order logics, and assume that the formulas are interpreted modulo some background theory $\mathcal{T}$ such as linear arithmetics, theory of bit vectors, or their combinations (see e.g. [7] and references therein). Like, for instance, in the SMT-LIB standard [8], we assume typed (i.e., sorted) logics. An *interpretation* $\mathcal{I}$ for a formula ϕ associates each variable in ϕ with an element in the domain of its type (e.g., an integer). Models and satisfiability are defined as usual.

1550-4808/11 $26.00 © 2011 IEEE
DOI 10.1109/ACSD.2011.29




```
MODULE TIMER(in, DELAY)
  VAR clk : clock(!in & next(in)); -- reset on a rising edge
  VAR out : boolean;
  INIT  !out
  TRANS next(out) = case
         !in & next(in)      : TRUE;  -- true on a rising edge
         out & clk < DELAY   : TRUE;  -- true for 3 seconds
         TRUE                : FALSE; -- otherwise
      esac;
  INVAR out -> (clk <= DELAY) -- must expire after 3 seconds
```

Figure 1. A part of a SIS (from [4]) and a NuSMV+clocks module for non-resettable timers such as T_2.

To model clocks, in the following we assume that the applied theory includes the standard theory of linear arithmetics over reals. We say that a variable x is a *clock variable* if (i) its type is real and, (ii) in all the formulas we consider, all the atoms involving it are of the form $c \bowtie j$, where $\bowtie \in \{<, \leq, =, \geq, >\}$ and $j \in \mathbb{Z}$. Observe that, as usual in timed automata context as well, one could use rational constants in systems and then scale them to integers in a behavior and property preserving way.

An STTS (or simply a system) is a tuple

$$\langle X, C, \mathit{Init}, \mathit{Invar}, T, R, U \rangle$$

where:

- $X = \{x_1, ..., x_n\}$ is a finite set of finite domain *state variables*. We use $X' = \{x'_i \mid x_i \in X\}$ to denote the set of corresponding, similarly typed *next state variables*.
- $C = \{c_1, ..., c_m\}$ is a finite set of clock variables (or simply *clocks*), disjoint from X.
- Init is a formula over X describing the initial states of the system.
- Invar is a formula over $X \cup C$ specifying an invariant.
- T is the *transition relation formula* over $X \cup C \cup X'$.
- R is a function that associates each clock $c \in C$ a *reset condition* formula r_c over $X \cup C \cup X'$.
- U is a formula over X stating when the system is in an *urgent state* and does not permit time elapse steps.

A system is *untimed* if it does not have any clock variables.

The semantics of an STTS is defined by its states and how they may evolve to others. A *state* is simply an interpretation over $X \cup C$. A state s is *valid* if it respects the invariant, i.e. $s(\mathit{Invar}) = \mathbf{t}$. A state s is an *initial state* if it is valid, $s(\mathit{Init}) = \mathbf{t}$, and $s(c) = 0$ for each clock $c \in C$. Given a state s and a $\delta \in \mathbb{R}_+$, we denote by $s + \delta$ the state where clocks have increased by δ, i.e. $(s + \delta)(c) = s(c) + \delta$ for each clock $c \in C$ and $(s + \delta)(x) = s(x)$ for each state variable $x \in X$. A valid state s may evolve into a successor state u, denoted by $s \longrightarrow u$, if u is also valid and either of the following holds:

1) *Discrete step*: (i) the current and next state interpretations evaluate the transition relation to true, i.e. $\gamma(T) = \mathbf{t}$ where $\gamma(y) = s(y)$ when $y \in X \cup C$ and $\gamma(x') = u(x)$ when $x' \in X'$, and (ii) each clock either resets or keeps its value: for each clock $c \in C$, $u(c) = 0$ if $\gamma(r_c) = \mathbf{t}$ and $u(c) = s(c)$ otherwise.
2) *Time elapse step*: (i) the system must not be in an urgent state: $s(U) = \mathbf{f}$, (ii) the time elapses some amount: $u = s + \delta$ for some $\delta \in \mathbb{R}_+$, and (iii) the invariant is respected in the states in between: $s + \mu$ is valid for all $0 < \mu \leq \delta$.

An *path* of the system is a finite or an infinite sequence $s_0 s_1 ...$ of states such that $s_i \longrightarrow s_{i+1}$ holds for each consecutive pair of states in the path. The urgent states are convenient in modeling atomic computation sequences (cf. use of urgent states in Uppaal [3]).

Example 1: Consider an STTS modeling the timer T_2 in the system in Fig. 1. The STTS has the clock variable T_2 which is reset when a discrete step makes the signal a true, i.e. $r_c = (\neg a \wedge a')$, corresponding to the activation of the timer. The output signal b is initially false, i.e. Init contains the conjunct $(\neg b)$. It changes to true when the signal a does and then stays true for three seconds. These properties are captured by the conjunct $(b' \Leftrightarrow (\neg a \wedge a') \vee (b \wedge (T_2 < 3)))$ in the transition relation T. To force the timer output to be reset after three seconds, Invar contains the conjunct $(b \Rightarrow (T_2 \leq 3))$. Given a state $\{b \mapsto \mathbf{f}, a \mapsto \mathbf{f}, T_2 \mapsto 7.9, ...\}$ of the system, a discrete step could change it to $\{b \mapsto \mathbf{t}, a \mapsto \mathbf{t}, T_2 \mapsto 0, ...\}$ and then a time elapse step could further lead to state $\{b \mapsto \mathbf{t}, a \mapsto \mathbf{t}, T_2 \mapsto 2.9, ...\}$.

A. Translating Timed Automata into STTS

Although STTSs do not have automata structure, they are basically generic symbolic presentations of transition systems augmented with clocks. This fact allows timed automata (see e.g. [9], [3]) to be efficiently translated into STTSs by capturing the automata structure with appropriately constrained finite-domain variables in STTSs. As an example, consider a part of a timed automata modeling Fischer's mutual exclusion protocol shown in Fig. 2. A simple translation of the automaton into an STTS could model the global finite-domain variable id with a state variable id of the same type in the STTS. To capture the automata structure, the translation could create, for instance, a state variable loc with finite domain $\{\text{idle}, \text{req}, ...\}$ for the control locations of the automaton and a clock variable

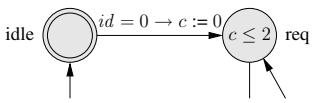


Figure 2. A part of a timed automata for Fischer's protocol

c for the clock c. Now the initial state condition $Init$ in STTS is the conjunction $(loc = \text{idle}) \wedge (id = 0)$, the transition relation formula T is the disjunction of translations of all transitions in the automata, including the disjunct $(loc = \text{idle}) \wedge (id = 0) \wedge (loc' = \text{req}) \wedge (id' = id)$ for the one shown in the figure, the reset condition r_c includes the conjunct $(loc = \text{idle}) \wedge (loc' = \text{req})$ and the invariant includes the conjunct $(loc = \text{req}) \Rightarrow (c \leq 2)$. For handling multiple transitions between locations and compositions of timed automata communicating via synchronization labels, see e.g. the symbolic translations in [10], [11]. One can translate both "urgent" and "committed" locations used in Uppaal [3] by using urgent state formulas in STTS.

In theory it is also possible to map an STTS with finite-domain state variables into a timed automaton by having an automaton location for each possible interpretation over the state variables and then setting the automaton transitions and location invariants accordingly. Of course, this is a quite impractical construct as there are exponentially many interpretations with respect to the number of STTS state variables. The possible urgency constraints in an STTS could be transformed into urgent locations if the target automata class supports them (e.g. Uppaal includes such support [3]); otherwise, additional clocks could be used to model them.

B. STTS as an extension to the NuSMV input language

One possible way for describing STTSs in practice is to extend the input language of an existing symbolic model checking tool, such as NuSMV [2], to support clock variables. An advantage of this approach is that existing NuSMV features can be used: NuSMV allows, for instance, modular description of systems.

We have devised such an extended NuSMV format (call it NuSMV+clocks). In it, clock variables can be described in addition to normal NuSMV variables. Each clock variable receives a reset condition as declaration argument. Clock variables can be used in expressions as long as the restrictions described in Sect. II are obeyed, i.e., the only operation allowed on clock variables is comparison against integer constants. The usual NuSMV constraint definitions can be used to describe invariant conjuncts ("`INVAR`"), initial constraints ("`INIT`" and "`ASSIGN`"), and transition relation ("`TRANS`" and "`ASSIGN`") of an STTS. An additional keyword `URGENT` is introduced to define urgency constraints.

As an example, the right hand side of Fig. 1 shows a NuSMV+clocks module for non-resettable timers such as T_2 in the SIS on the left hand side. To implement T_2, one only needs to instantiate the module, e.g. with

```
VAR T2 : Timer(a, 3);
DEFINE b = T2.out;
```

III. Translating STTS to untimed systems

In this section we describe our first method for symbolic model checking of STTSs. It is based on encoding region abstraction [9] in a symbolic form using propositional logic. After this, provided that the non-clock state variables are finite-domain and thus booleanizable, one can apply, for instance, standard BDD-based model checking techniques (see, e.g., [1]) to verify properties of the original STTS. Our basic encoding is based on the timed-automaton-to-Kripke-structure translation of [6]. Similar timed-to-untimed-system translation approaches have been evaluated in the context of bounded model checking, e.g. [12], [13]. We also describe an extension to the basic encoding that aims at accelerating symbolic model checking by allowing multiple regions to be reached in one step.

A. Region abstraction

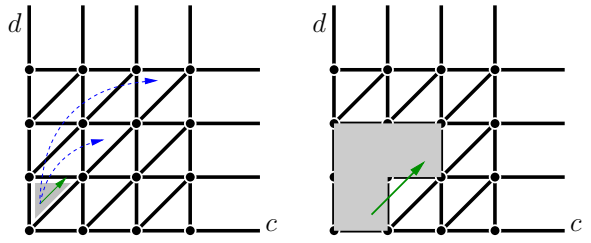
A key challenge in translating an STTS into an untimed system is mapping the infinite number of clock valuations of the STTS to the finite state space of an untimed system. In our approach, we use the region abstraction [9] to solve this problem. Similar to the construction of the region automaton in [9] or the timed-automaton-to-Kripke-structure translation described in [6], we map a state in the original STTS to a combination of the valuation of the state variables and a region of equivalent clock values in the untimed system.

Assume an STTS $S = \langle X, C, Init, Invar, T, R, U\rangle$. Let v be an interpretation over C (also called a *clock valuation*). For each clock $c \in C$, let $\lfloor v(c)\rfloor$ and $\langle v(c)\rangle$ be its integer and fractional parts, respectively, i.e. $v(c) = \lfloor v(c)\rfloor + \langle v(c)\rangle$ with $\lfloor v(c)\rfloor \in \mathbb{N}$ and $0 \leq \langle v(c)\rangle < 1$. Furthermore, let m_c be c's maximum (relevant) value, i.e. the largest constant that c is compared to in $Invar$, T or R. Two clock valuations v and w belong to the same *region*, denoted by $v \sim w$, if

1) for all clocks $c \in C$ either $\lfloor v(c)\rfloor = \lfloor w(c)\rfloor$ or $v(c) > m_c$ and $w(c) > m_c$
2) for all clocks $c, d \in C$ with $v(c) \leq m_c$ and $v(d) \leq m_d$, it holds that $\langle v(c)\rangle \leq \langle v(d)\rangle$ iff $\langle w(c)\rangle \leq \langle w(d)\rangle$.
3) for all clocks $c \in C$ with $v(c) \leq m_c$, it holds that $\langle v(c)\rangle = 0$ iff $\langle w(c)\rangle = 0$

Figure 3(a) illustrates the region abstraction for an STTS with two clocks, c and d, and $m_c = m_d = 3$. The thick black lines, thick black dots and the areas in between the thick black lines each represent a different region (please ignore the arrows for now). The region in which $\lfloor v(c)\rfloor = \lfloor v(d)\rfloor = 0$ and $0 < \langle v(c)\rangle < \langle v(d)\rangle$ is highlighted in gray.

Intuitively, the clock valuations inside a clock region are equivalent in two ways. Firstly, each clock expression in an STTS is either (i) true for all valuations in a given region, or (ii) false for all valuations in the region. This is because clock variables inside an STTS can only be compared to



(a) Regions (b) Non-convex invariant

Figure 3. Region abstraction and a non-convex invariant for a system with two clocks

integers. Secondly, for any path starting from a given state in the STTS, a similar (i.e. one visiting the same clock regions) path starts from any clock valuation in the same region.

Any given region in the region abstraction has a unique *successor region*. Usually, this is the next region one reaches by starting at a point inside the current region and letting time pass. An exception is the region in which all clocks have exceeded their maximum value – this region is by definition its own successor. In the example in Fig. 3(a), any region's successor region can be reached by moving diagonally up and right; e.g., the successor of the gray region is the region with $\lfloor v(c) \rfloor = 0$, $\lfloor v(d) \rfloor = 1$, $0 = \langle v(d) \rangle < \langle v(c) \rangle$ shown by the solid line arrow. More formally, the following rules define, how to reach any region's successor region.

1) No changes are made to clocks that already exceeded their maximum value.
2) If one or multiple clocks have a zero fractional part, some time passes so that no clock's fractional part is zero any more but also no clock reaches the next integer value yet. That is, afterwards all clocks' fractional parts are greater than zero while the ordering of their fractional parts and the values of their integer parts remain unchanged.
3) If no clock's fractional part is zero, then the clocks that have the highest fractional part reach their next integer value and the fractional parts of these clocks are reset to zero. The integer parts and the ordering of fractional parts of all other clocks remain unchanged.

B. Symbolic representation of clock regions

Assume an STTS $S = \langle X, C, Init, Invar, T, R, U \rangle$. The aim of the untiming process is to translate S into an equivalent untimed system $S_U = \langle X_U, \emptyset, Init_U, Invar_U, T_U, \emptyset, \mathbf{f} \rangle$ having no clock variables, reset conditions or urgency constraints. The transitions of such a system are defined in the same way as discrete steps in an STTS are defined.

As a first step in the translation of S into S_U, S's clocks are replaced by additional state variables representing the current clock region. For each clock $c \in C$, integer variables i_c and f_c are introduced, where i_c represents $\lfloor v(c) \rfloor$ while f_c describes the ordering between the fractional parts of clock values, i.e., for any two clocks $c, d \in C$, a region in which $\langle v(c) \rangle \leq \langle v(d) \rangle$ is represented by $f_c \leq f_d$. Furthermore, $f_c = 0$ iff $\langle v(c) \rangle = 0$ in the current region. The special case where $v(c) > m_c$ is represented by $i_c = m_c$ and $f_c = 1$. For any clock $c \in C$, the domain of i_c is $\{0, \ldots, m_c\}$ while f_c's domain depends on $|C|$ and the exact encoding used in the translation and will be discussed in Sect. III-E.

As described in Sect. II, an STTS can perform two different types of steps: time elapse steps and discrete steps. A state variable $elapsed$ is used in S_U to differentiate between these types, resulting in the following definition of X_U: $X_U = X \cup \{i_c, f_c \mid c \in C\} \cup \{elapsed\}$.

Counter-intuitively, $elapsed$ does not determine the type of the next step but the type of the most recent step. If $elapsed$ would determine the type of the next step, then S_U would have two different initial states, one from which all traces start with a time elapse step and one from which all traces start with a discrete step. In model checking tools like NuSMV, specifications are implicitly universally quantified over all initial states. This may lead to counter-intuitive situations. For instance, if a given property holds for some initial state but does not hold for another one, then both the property and its negation will be reported to not hold. To avoid such situations in S_U, $elapsed$ denotes the type of the previous step and therefore S_U has only one initial state, unless S also has more than one initial state.

As described in Sect. II, $Invar$, T and the values of the R function are logical formulas over S's state variables and clocks (in which clocks can only occur in comparisons with integer constants). In the translation of S to S_U these formulas have to be translated from formulas over clocks into formulas over clock regions. This is done by the function $untime$, which replaces clocks comparisons with formulas over the i_c and f_c variables. For instance, for a clock $c \in C$ and an integer constant j, it holds that $c > j$ iff $\lfloor c \rfloor > j$ or $\lfloor c \rfloor = j$ and $\langle c \rangle > 0$. Therefore, $untime(c > j) = i_c > j \vee (i_c = j \wedge f_c > 0)$. The full definition of the $untime$ function is given as follows, where $c \in C$ and $j \in \mathbb{N}$:

ϕ	$untime(\phi)$	ϕ	$untime(\phi)$
$c \leq j$	$i_c < j \vee (i_c = j \wedge f_c = 0)$	$c \geq j$	$i_c \geq j$
$c > j$	$i_c > j \vee (i_c = j \wedge f_c > 0)$	$c < j$	$i_c < j$
$c = j$	$i_c = j \wedge f_c = 0$		
Any other formula: Recursively untime subformulas			

C. Initial constraint, invariant and transition relation

A state is a valid initial state of S if its state variable interpretation satisfies $Init$ and all clock variables are zero. These properties are reflected by the following definition of the initial constraint of S_U:

$$Init_U := Init \wedge \bigwedge_{c \in C} untime(c = 0)$$

Unlike the initial constraint, the invariant of an STTS can contain clock variables and therefore has to be untimed before being used in S_U:

$$Invar_U := untime(Invar)$$

While the $Init_U$ and $Invar_U$ are just slightly modified versions of $Init$ and $Invar$, T_U does not only reflect T but also contains constraints that define how S_U moves between clock regions. As a results, the definition of T_U is much more complex. For clarity T_U is thus defined in two parts,

$$T_U := T_d \wedge T_e$$

where T_d ensures that discrete steps are executed correctly while T_e takes care of time elapse steps.

The transition constraints for discrete steps, T_d, are closely related to the original timed transition relation T. In a discrete step, the combination of new and old values of the state variables and the current clock valuation has to satisfy T. In addition, all clock values remain unchanged, unless they are reset. T_d results from joining these constraints:

$$\begin{aligned} T_d := \neg elapsed' \Rightarrow untime(T) \wedge \\ \bigwedge_{c \in C} ((untime(r_c) \Rightarrow (i'_c = 0 \wedge f'_c = 0)) \wedge \\ (\neg untime(r_c) \Rightarrow (i'_c = i_c \wedge f'_c = f_c))) \end{aligned}$$

In a time elapse step, in contrast, the state variables remain unchanged while the clock variables are updated to represent the successor clock region. Furthermore, the urgency constraint U may not hold as this would forbid time elapse steps. The constraints for updating the clock variables are described separately in Sects. III-D and III-E, and are here referred to as $T_{e,i}$ and $T_{e,f}$. We define T_e by

$$T_e := elapsed' \Rightarrow \neg U \wedge \bigwedge_{x \in X} (x' = x) \wedge T_{e,i} \wedge T_{e,f}.$$

D. Updating the clocks' integral variables

In any time elapse step in the untimed systems S_U, the clock variables are updated so that they represent the successor of the current region. $T_{e,i}$ and $T_{e,f}$ are the parts of the untimed system's transition relation T_U that ensure that the clock variables are updated according to the rules for reaching the successor region given in Sect. III-A. More precisely, $T_{e,i}$ contains the rules for updating the clocks' integral variables and $T_{e,f}$ contains rules for updating the clocks' fractional variables. We define $T_{e,i}$ as

$$\begin{aligned} T_{e,i} := \bigwedge_{c \in C} ((next\_int_c \Rightarrow i'_c = i_c + 1) \wedge \\ (\neg next\_int_c \Rightarrow i'_c = i_c)) \end{aligned} \quad \text{(III.1)}$$

where $next\_int_c$ is defined as follows:

$$next\_int_c := i_c < m_c \wedge \bigwedge_{d \in C} (0 < f_d \wedge f_d \leq f_c) \quad \text{(III.2)}$$

The shorthand $next\_int_c$ defines, under which circumstances i_c is increased. Firstly, corresponding to the first rule from Sect. III-A, i_c is only increased until c reaches m_c. Secondly, the subformula $\bigwedge_{d \in C} 0 < f_d \leq f_c$ summarizes the second and third rule for determining the successor region. The formula is false, if any clock's fractional part is currently zero. In that case, according to the second rule, no clock reaches the next integer value. The formula is also false, if any other clock has a greater fractional part than c, as in this case the respective clock will reach the next integer value first. Formula III.1 defines that i_c is increased if $next\_int_c$ holds and remains unchanged otherwise.

E. Updating the clocks' fractional variables

The formula $T_{e,f}$ defines how the clocks' fractional variables are updated on time elapse steps. Like $T_{e,i}$, $T_{e,f}$ is based on the rules for finding successor regions given in Sect. III-A. Two alternative encodings are given for $T_{e,f}$. The first encoding, called the *implicit encoding*, is based on specifying the conditions under which the ordering of the fractional variables' values may or may not change. The second encoding explicitly defines the value of each fractional variable after a time elapse transition and therefore is called the *explicit encoding*.

1) The implicit encoding: The implicit encoding is based on specifying under which conditions the ordering of the fractional variables may change on a time elapse step. Here $T_{e,f}$ is defined by

$$\begin{aligned} T_{e,f} := \bigwedge_{c \in C} ((f'_c = 0 \Leftrightarrow next\_int_c) \wedge (i_c = m_c \Rightarrow f'_c = 1) \wedge \\ \bigwedge_{d \in C} ((i_c \neq m_c \wedge i_d \neq m_d \wedge f'_c \neq 0 \wedge f'_d \neq 0) \Rightarrow \\ (f_c \leq f_d \Leftrightarrow f'_c \leq f'_d))) \end{aligned} \quad \text{(III.3)}$$

where $next\_int_c$ is defined as in Sect. III-D.

The first part of the formula ensures two things. Firstly, after a time elapse step f_c is zero iff c reaches the next integer value in the current step. This ensures both that (i) f_c is reset whenever the clock c reaches the next integer value and (ii) f_c is not equal to zero after a time elapse step otherwise, corresponding to the second rule in Sect. III-A. Secondly, f_c will stay one once c exceeds its maximum value (one could only require that the value is greater than zero; this would however require modification of $next\_int_c$ to ensure that clocks that exceeded their maximum value are not taken into consideration when checking whether another clock reaches its next integral value in the next step).

The second part of the formula enforces the following: any two clocks for which none of the mentioned special cases applies, i.e. they have not exceeded their maximum value and do not reach the next integer value, should keep the ordering of their fractional variables.

For the implicit encoding the domain for the fractional variables can be set to $\{0, \ldots, |C|\}$ so that all of them can have different non-zero values.

2) The explicit encoding: Similar to the definition of $T_{e,i}$, the explicit encoding explicitly defines the value of f'_c for time elapse steps based on the current values of the clock variables. In the explicit encoding, $T_{e,f}$ is defined as follows:

$$\begin{aligned} T_{e,f} \coloneqq \bigwedge_{c \in C} & (i_c = m_c \Rightarrow f'_c = 1) \wedge \\ & (i_c < m_c \wedge next\_int_c \Rightarrow f'_c = 0) \wedge \\ & (i_c < m_c \wedge \neg next\_int_c \wedge inc_c \Rightarrow f'_c = f_c + 1) \wedge \\ & (i_c < m_c \wedge \neg next\_int_c \wedge \neg inc_c \Rightarrow f'_c = f_c) \end{aligned}$$

where $next\_int_c$ is defined as in Sect. III-D and

$$inc_c \coloneqq f_c = 0 \vee \bigvee_{d \in C \setminus \{c\}} f_d = f_c - 1 \qquad \text{(III.4)}$$

Again, the definition of $T_{e,f}$ is based on the rules for reaching the next clock region given in Sect. III-D. Corresponding to the first of these rules, the first line in the definition of $T_{e,f}$ ensures that f_c stays one once c reaches m_c. The second line ensures that f_c is reset to zero whenever c reaches the next integer value. Corresponding to rule two, the third line of $T_{e,f}$ ensures that the next value of f_c is greater than zero if it's current value is zero. Furthermore, the third and fourth lines ensure that, except for clocks that exceeded their maximum value or reach the next integer value, the ordering of the values of the f_d variables remains unchanged. If there is a clock d such that $f_d = f_c - 1$, then there is a possibility that f_d is increased. In such a situation inc_c is true and, in order to keep the ordering of f_c and f_d, f_c is increased. Otherwise, f_c stays the same.

When using the explicit encoding, the value of any fractional variable can reach but not exceed $2*|C|+1$. Therefore, the domain of any fractional variable is $\{0, \ldots, 2 * |C| + 1\}$.

F. Verifying properties on STTS

The untimed system S_U allows to verify properties on the original system S. Verifying invariant specifications on S is fairly straightforward. Let ϕ be an invariant specification, i.e. a formula over $X \cup C$. Then ϕ holds globally in S iff $untime(\phi)$ holds globally in S_U. More complicated properties can be checked by checking properties on S_U in other logics like CTL and LTL [1]. As long as neither any clocks nor the next-time operator $\mathbf{X}$ are used in such properties, checking a property on S_U corresponds to checking the property on S, which could be proven based on the observation that S_U is (restricting to the non-clock variables) stutteringly equivalent [14] to S. The proof for the stuttering equivalence would be similar to the proof of the bisimilarity of a timed automaton and it's region automaton in [9].

For verifying properties specified with the real time temporal logic TCTL [15], techniques proposed in [6] could perhaps be used; this issue is left as future work.

In the verification of timed systems, it is often desirable to exclude zeno paths, i.e. paths on which time does not diverge. In a untimed system resulting from translating an STTS they could be excluded using fairness constraints.

G. Time leaps

In the encodings above, a time elapse step in the translated system S_U lets time advance only to the immediate successor clock region. Traversing the regions this way can be fairly slow, i.e., a large number of time elapse steps is needed to reach regions in which the clocks have high values. This leads to a large diameter of the state space of S_U, which may slow down the verification. As a potential solution to this issue, we propose to extend the encodings with *time leaps*. Unlike normal time elapse steps, time leaps do not lead to the successor region but to a region in the future having the same ordering of clocks' fractional parts. This corresponds to time advancing by a number of full time units. As an example, the dashed arrows in Fig. 3(a) illustrate the two time leaps one can take from the gray region (dashed arrows). Compared to one time leap, eight normal time elapse steps are needed to reach the farther of these regions from the gray region.

Time leaps are not intended to alter the semantics of S_U but merely to allow shortcuts that reduce the number of time elapse steps needed to reach a certain clock region. That is, for any taken time leap the equivalent sequence of time elapse steps should be feasible as well. To ensure this, one has to guarantee that every intermediate state a time leap jumps over satisfies the invariant of S.

A key property of invariants in this respect is convexity. Intuitively, an invariant $Invar$ is convex if it cannot first become false and then true again as time passes. Formally, $Invar$ is convex if for each state and each $\delta_2 > \delta > 0$ it holds that $(Invar \wedge Invar_{\delta_2}) \Rightarrow Invar_\delta$, where $Invar_\delta$ is $Invar$ with any occurrence of any clock $c \in C$ replaced with $c+\delta$. As an example, the gray area shown in Fig. 3(b) is not convex. Whether or not a given invariant $Invar$ is convex can be determined by checking whether $Invar \wedge \neg Invar_\delta \wedge Invar_{\delta_2} \wedge 0 < \delta < \delta_2$ is satisfiable using an SMT-solver (e.g., Yices [16]). If an invariant is convex, then it is impossible to leap over a point in time where the it does not hold to a point where it holds again. Hence, convex invariants do not need special consideration with respect to time leaps.

If an invariant $Invar$ is not convex, then certain time leaps have to be forbidden to avoid leaping over states that violate $Invar$. A way to enforce this is to ensure that the truth value of no subexpression of $Invar$ changes along a leap. Let $\phi = c \bowtie j$ be a subexpression of $Invar$ with c being a clock and j an integer constant. If $\bowtie \in \{<, \leq, \geq, >\}$, then both ϕ and $\neg\phi$ are convex. Therefore, if ϕ has the same truth value both at the starting state and the end state of a given time leap, then ϕ is guaranteed to have the same truth value in any intermediate state that has been leaped over. Furthermore, if all subexpressions of $Invar$ of the described form have the

same truth value at the start and the end of a time leap, then $Invar$ does not change its truth value in any intermediate state that has been leaped over, assuming $Invar$ does not contain any clock constraints that check for equality. Thus it is possible to ensure that a time leap does not leap over a state violating $Invar$ by adding, for every clock comparison sub-expression $c \bowtie j$ in $Invar$, the conjunct

$$leap \Rightarrow (untime(c \bowtie j) \Leftrightarrow untime(c \bowtie j)')$$

to the transition relation of the untimed system, where $leap$ denotes that the current step is a time leap. Unfortunately, this approach does not work for clock comparisons of the form $c = j$ as their negation is not convex. As a result, they can change from false to true and back to false as time passes. It is however, possible to first replace $c = j$ with $c \leq j \wedge c \geq j$ and to then proceed as described. A small optimization to the approach could be to split up invariants that consist of conjunctions into their conjuncts and then to check each conjunct for convexity and add the constraints restricting time leaps only for the non-convex conjuncts.

There are several possible ways to implement time leaps. Basically, one modifies the formulas in Sects. III-D and III-E to allow the integer parts to increase non-deterministically by the same amount and, when this happens, forces the fractional parts to stay the same. Due to lack of space, we omit the rather tedious exact definitions and possible encoding variants here.

Time leaps do not affect the set of reachable states in S_U. Thus invariant specifications do not need any special considerations when using time leaps. Similarly, the set of paths in S_U remains, except for the possibility of using shortcuts, the same. Thus, LTL or CTL specifications do not need any special consideration as long as they do not reference any clock variables or use the next-time operator.

IV. SMT-BASED BOUNDED MODEL CHECKING

In addition to the untiming approaches presented above, we consider SMT-based bounded model checking (BMC) of STTSs. In BMC [17] the basic idea is to build, given a *bound* $k \in \mathbb{N}$, a formula that is satisfiable if and only if the system has an path $s_0 s_1 ... s_k$ of k steps that violates a given property ϕ. The satisfiability of the formula is then checked with a satisfiability solver. If the formula is satisfiable, then any model of it represents a counter-example to the property, illustrating how the property can be violated and helping in debugging; if the formula is unsatisfiable, then the process is repeated with a larger bound until a counter-example is found, the diameter of the system is reached or one runs out of resources (time or memory).

Here, we describe how bounded model checking of invariance properties can be applied to STTS using an SMT solver supporting the very simple fragment of linear real arithmetics involving equality and comparison with constants. The given BMC encoding of STTSs is considered only a proof of concept. In particular, it only works for STTS that have a convex (cf. Sect. III-G) invariant. In fact, it is not easy to see how non-convexity could be handled in BMC. Furthermore, it would be relatively straightforward to extend the BMC encoding (i) with the ability to check more complex properties expressed in linear time temporal logic and (ii) to exploit incremental solving interfaces of modern SMT solvers; for these techniques, see e.g. [18] and the references therein. The encoding we present is conceptually somewhat simpler than the ones for timed automata, e.g., in [10], [11] as we do not have to take care of the automata structure and synchronization between multiple automata.

Assume a system $S = \langle X, C, Init, Invar, T, R, U \rangle$. Let ϕ be an invariance property, i.e. a formula over $X \cup C$, that we wish to verify to hold in all states reachable from any initial state of the system. Given an $i \in \mathbb{N}$, let $X^{[i]} = \{x^{[i]} \mid x \in X\}$ be a "timed copy" of the state variables, $x^{[i]}$ representing the state variable x at the ith state in a bounded path $s_0 s_1 ... s_k$. Define $C^{[i]}$ similarly. In addition to the state and clock variables, k boolean variables $e^{[0]}, \ldots, e^{[k-1]}$ are used to determine the type of the transitions in the path: $e^{[i]}$ is true in a model iff the step from state i to state $i+1$ is a time elapse step and false if the step is a discrete step. Furthermore, k real-valued variables $\delta^{[0]}, \ldots, \delta^{[k-1]}$ denote the amount of time passed in each time elapse step.

Now given a bound $k \geq 0$, we form a formula $|[S, \phi, k]|$ which is satisfiable if and only if S has an path of length k in which at least one state of the states violates the property ϕ. That formula is defined by

$$|[S, \phi, k]| := |[Init]| \wedge |[Invar, k]| \wedge |[T, R, U, k]| \wedge |[\phi, k]|$$

where the part $|[Init]| := Init(X^{[0]}) \wedge \bigwedge_{c \in C}(c^{[0]}{=}0)$ ensures that $X^{[0]} \cup C^{[0]}$ represents an initial state, $|[Invar, k]| := \bigwedge_{0 \leq i \leq k} Invar(X^{[i]}, C^{[i]})$ forces all the $k{+}1$ states to respect invariants, $|[\phi, k]| := \bigvee_{0 \leq i \leq k} \neg\phi(X^{[i]}, C^{[i]})$ requires at least one of the states to violate the property, and $|[T, Invar, U, k]|$ ensures that the semantics of STTS are respected. The formula $|[T, R, U, k]|$ is the conjunction of (i) the formula

$$\bigwedge_{0 \leq i < k} \Big(e^{[i]} \Rightarrow \neg U(X^{[i]}) \wedge (\delta^{[i]} \geq 0) \wedge \bigwedge_{c \in C} (c^{[i+1]} = c^{[i]} + \delta^{[i]}) \wedge \bigwedge_{x \in X} (x^{[i+1]}{=}x^{[i]}) \Big)$$

stating that if a time elapse steps is taken, then the state is not urgent, all the clocks are updated, and the state variable values stay the same, and (ii) the formula

$$\bigwedge_{0 \leq i < k} \Big(\neg e^{[i]} \Rightarrow T(X^{[i]}, C^{[i]}, X^{[i+1]}) \wedge \bigwedge_{c \in C} (r_c(X^{[i]}, C^{[i]}, X^{[i+1]}) \Rightarrow (c^{[i+1]}{=}0)) \wedge \bigwedge_{c \in C} (\neg r_c(X^{[i]}, C^{[i]}, X^{[i+1]}) \Rightarrow (c^{[i+1]}{=}c^{[i]})) \Big)$$

making the discrete steps to respect the transition relation formula and the clock reset conditions. Any interpretation satisfying $|[S, \phi, k]|$ represents a k-step path on which at least one state violates the invariant specification ϕ.

V. Experimental Results

To evaluate the proposed techniques in practice, we apply them to an industrial SIS as well as to synthetic benchmarks using the untiming-based verification approach described in Sect. III and the BMC encoding in Sect. IV.

A. The sensors benchmark

To evaluate the scalability of our verification approaches, we developed a simple, scalable synthetic SIS. The synthetic SIS consists of a number of sensors and alarms. Each sensor nondeterministically increases or decreases its value. Each alarm is connected to multiple sensors. For every connected sensor, an alarm observes whether or not the sensor is inside a certain critical range. If all sensors' values remain inside their respective critical region for a certain amount of time, the alarm is activated. We encoded the sensors benchmark in the extended NuSMV file format described in Sect. II-B.

The sensors benchmark has a variety of parameters that can be modified, e.g. the sensor value's range or the time after which an alarm is activated. We fixed all parameters except for the number of sensors and alarms. Increasing the number of sensors increases the number of state variables in the STTS while a higher number of alarms leads to a larger number of clocks. Therefore, varying the numbers of alarms and sensors allows evaluate how a verification approach scales in the number of state variables and clocks.

We fixed the sensor values to range from zero to 100 and to change by at most ten units in each step. Each alarm is connected to all but one sensor and has a separate, ten units wide critical range for every sensor. The critical ranges were randomly generated in advance, so that for a given pair of a sensor and an alarm the same critical range is used in every one of the different settings used. The delay of the first alarm is 100, i.e. the alarm is activated, if the connected sensors stay inside their critical ranges for 100 time units. The delay of the second alarm is 101, the one for the third 102 and so forth. Note that using the same delay for all alarms would allow to scale time without loosing behaviors and therefore would reduce the number of regions significantly. CTL specifications stating that each alarm individually can reach an active state but not all alarms can be active at once were added. The critical sensor ranges were generated in a way that ensures that the latter specification holds.

Using the sensors benchmark, different configurations for the translation described in Sect. III were compared. Both the implicit encoding and the explicit encoding were tried. Each setting was tried with and without time leaps. Non-zenoness of potential counter-examples was enforced.

First, we study how well the different translations scale in the number of clocks in the STTS by letting the number of alarms range from two to ten and fixing the number of sensors to five. Each untimed test system was checked using NuSMV [2] version 2.5.2. Apart from activating dynamic BDD reordering, NuSMV was used with default settings. All experiments were executed on a Linux computer with a 2.33GHz Intel Core 2 Duo E6550 processor with memory usage limited to two gigabytes and CPU time to one hour.

Table I shows verification times for different translations and different numbers of alarms. The times do not include the time needed for untiming, which was, however, negligible (at most 0.2 seconds). The explicit encoding performed significantly better than the implicit encoding. With time leaps enabled, systems with up to seven clocks could be verified. For both encodings, using time leaps significantly reduced the time required for verification. Note, however, that the effect can be expected to be much weaker if the clocks have lower maximum values.

In a second experiment, the sensors benchmark was used to compare performance and scalability of our translation-based verification approach to the real-time model checker Uppaal [3]. For this purpose, Uppaal versions of the benchmark were implemented and verified using the Uppaal command line tool version 4.0.11. Uppaal was used with depth-first search, which for the sensors benchmark is slightly faster than breadth-first, and otherwise default settings. Our translation was used with the explicit encoding, as it had been found to perform better in the first experiment.

Table II shows the verification times for the sensors benchmark using Uppaal and our translation-based approach with and without time leaps Again, time leaps reduced the time required significantly. Unsurprisingly, Uppaal scaled very well in the number of clocks used. When using only two sensors, Uppaal was easily able to handle ten clocks and very likely would have been able to handle even larger numbers. High numbers of sensors, however, are problematic for Uppaal. A likely reason are the large non-time related component of the state space of the sensors benchmark and its high amount of non-determinism. The BDD-based verification techniques used by NuSMV, in contrast, handle the large state space and the non-determinism very well but do not scale as well in the number of clocks. As a result, the combination of our translation and NuSMV scaled better in the number of sensors and worse in the number of alarms. Surprisingly, NuSMV consistently performed worse with a low number of sensors than with medium numbers of sensors. A likely source of this effect is the unpredictable nature of verification times with BDD-based methods.

B. Real world benchmark

To evaluate our approaches in an industrial setting, we applied them to an SIS modeling an industrial safety system. We used both the translation described in Sect. III and the

Table I
COMPARISON OF VERIFICATIONS TIMES (CPU TIME IN SECONDS) FOR FIVE SENSORS AND A VARYING NUMBER OF ALARMS.

		2 alarms	3 alarms	4 alarms	5 alarms	6 alarms	7 alarms	8 alarms
Explicit encoding	No time leaps	28.35	85.75	122.63	558.54	1214.16	*timeout*	
Explicit encoding	Time leaps	**5.09**	**7.94**	**19.94**	**46.32**	**430.01**	**896.79**	*timeout*
Implicit encoding	No time leaps	29.33	651.24	1104.2	1540.26	*timeout*		
Implicit encoding	Time leaps	17.67	81.34	72.67	572.44	*timeout*		

Table II
COMPARISON OF VERIFICATION TIMES (CPU TIME IN SECONDS) FOR VARYING NUMBERS OF SENSORS AND ALARMS USING NUSMV AND UPPAAL.

(a) Untime + NuSMV, explicit encoding, time leaps disabled

Sensors \ Alarms	2	3	4	5	6	7
2	8.7	174.1	*timeout*			
3	8.3	28.7	201.0	3253.5	*timeout*	
4	12.6	108.6	122.1	1752.5	2125.7	*timeout*
5	28.4	85.8	122.6	558.5	1214.2	*timeout*
6	38.6	478.1	1097.8	671.2	2325.9	*timeout*
7	40.1	1346.4	348.8	843.8	2563.3	*timeout*
8	79.8	225.4	440.6	1751.6	2151.9	*timeout*
9	114.8	*timeout*				
10	1192.39	*timeout*				

(b) Untime + NuSMV, explicit encoding, time leaps enabled

Sensors \ Alarms	2	3	4	5	6	7	8
2	1.2	7.9	*timeout*				
3	**1.2**	**2.0**	**38.0**	**141.7**	**884.5**	*timeout*	
4	**2.8**	**4.6**	**14.1**	**45.2**	**805.3**	**1508.1**	*timeout*
5	**5.1**	**8.0**	**20.0**	**46.3**	**430.0**	**896.8**	*timeout*
6	**11.4**	**17.4**	**38.5**	**124.0**	**346.1**	**1353.6**	*timeout*
7	**17.0**	**25.1**	**51.5**	**274.1**	**783.6**	**1345.1**	*timeout*
8	**39.4**	**54.7**	**63.8**	**435.7**	**2101.0**	*timeout*	
9	**43.7**	**117.6**	**1095.1**	**658.6**	*timeout*		
10	**161.4**	**453.0**	**791.7**	**1823.5**	*timeout*		

(c) Uppaal, depth-first search

Sensors \ Alarms	2	3	4	5	6	7	8	9	10
2	**0.03**	**2.3**	**5.9**	**9.2**	**13.7**	**20.2**	**33.2**	**45.6**	**64.1**
3	230.5	341.5	563.5	761.5	1517.9	**2213.1**	*timeout*	*timeout*	*timeout*
4	*timeout*	*timeout*	*timeout*	*timeout*	*timeout*	*timeout*	*timeout*		

BMC encoding described in Sect. IV. The SIS was given in the form of a functional block diagram consisting of 27 functional blocks, e.g. flip-flops or delay elements. Ten blocks use clocks. In addition to its actual functionality, each block is aware of and reacts to faulty inputs.

Nine properties were given for the system, five of which hold; all were given as LTL formulas although six of them just specify invariants. Previously, the system has been verified using a discrete time model. However, using discretized time only considers an under-approximation of the actual behaviors of a system. In our attempts to verify the system using Uppaal, we had difficulties to model the system due to its logic-based nature and large number of nondeterministic inputs. Furthermore, we failed to verify the system using Uppaal within the available computing time and memory.

We modeled the SIS in the input language described in Sect. II-B and then tried to verify the properties using the translation described in Sect. III. We used the configuration that had performed best in the synthetic benchmarks, i.e. the implicit encoding with time leaps enabled. The same computer and NuSMV configuration as for the sensors benchmarks were used, except that cone-of-influence reduction was activated and counter-example computation disabled this time. The specifications were checked using LTL-model checking, even the ones that could have been turned into invariant-specifications. Non-zenoness of potential counter-examples was enforced. Unfortunately, not a single property could be verified using this setup. This result is not very surprising, considering that in the simpler sensors benchmark only systems with up to seven clocks could be verified.

The SIS has seven outputs. None of the properties references all of these outputs. Therefore, in a second step, we generated for each property a reduced SIS that only contains the functional blocks that are connected directly or indirectly to the outputs used in the property. On the reduced systems, two out of the nine properties could be verified. Table III shows the time needed for the verification.

In addition to using NuSMV for verifying the untimed SIS, we also used the BMC encoding introduced in Sect. IV. As our prototype implementation currently only supports invariant specifications, only six specifications could be checked this way. We checked these properties with bounds ten, twenty and forty. For the properties that had been known to not hold, counter-examples were already found with bound ten. For the properties that hold, no counter-examples can be found. Table III shows the times required for BMC using our encoding with the SMT-solver Yices [16].

VI. CONCLUSIONS

This paper introduces symbolic timed transition systems (STTS), a formalism for defining timed systems like SIS and two possible verification approaches for STTS. Firstly, a technique that is based on translating STTS to untimed systems that can be checked with classical BDD-based model checkers is proposed. Secondly, a SMT-based bounded model checking encoding is introduced.

Table III
VERIFICATION TIMES IN (CPU TIME IN SECONDS) FOR THE INDUSTRIAL BENCHMARK.

Property	Untiming-based		Bounded model checking			
	Full system	Reduced systems	Bound 10	Bound 20	Bound 40	Counter-example found?
0	*timeout*	*timeout*	*non-invariant property*			
1	*timeout*	*timeout*	*non-invariant property*			
2	*timeout*	*timeout*	1.02	5.49	1509.43	yes
3	*timeout*	*timeout*	0.67	1.46	6.33	no
4	*timeout*	*timeout*	*non-invariant property*			
5	*timeout*	9.74	0.76	3.25	330.51	yes
6	*timeout*	4.82	0.66	1.68	4.92	no
7	*timeout*	*timeout*	1.00	6.07	693.59	yes
8	*timeout*	*timeout*	0.67	1.40	6.67	no

A synthetic benchmark has been used to compare different configurations of the untiming-based verification approach and to evaluate how well it scales in the number of clocks and the size of the state space of the verified STTS. The scalability has been compared to the model checker Uppaal. Our approach was found to scale better in the size of the state space of the system but worse in its number of clocks.

An industrial system has been used to check the applicability of the approaches in practice. The untiming-based verification approach failed to verify most properties, which indicates that the investigation of possible improvements to its performance or alternative, more efficient complete methods for verifying STTS would be an interesting area for future research. Using bounded model checking, we were able to find counter-examples for some of the properties and show their absence up to a certain bound for others.

ACKNOWLEDGEMENT.

The financial support by the Academy of Finland (projects 128050 and 122399) is gratefully acknowledged.

REFERENCES

[1] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled, *Model Checking*. The MIT Press, 1999.

[2] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV 2: An opensource tool for symbolic model checking," in *Proc. CAV 2002*, ser. LNCS, vol. 2404. Springer, 2002, pp. 359–364.

[3] G. Behrmann, A. David, and K. G. Larsen, "A tutorial on UPPAAL," in *Proc. FM-RT 2004*, ser. LNCS, vol. 3185. Springer, September 2004, pp. 200–236.

[4] K. Björkmann, J. Frits, J. Valkonen, K. Heljanko, and I. Niemelä, "Model-based analysis of a stepwise shutdown logic," VTT Technical Research Centre of Finland, VTT Working Papers 115, 2009.

[5] J. Lahtinen, J. Valkonen, K. Björkman, J. Frits, and I. Niemelä, "Model checking methodology for supporting safety critical software development and verification," in *Reliability, Risk and Safety—Back to the Future*. CRC Press, 2010, pp. 2056–2063.

[6] T. A. Henzinger and O. Kupferman, "From quantity to quality," in *Proc. HART 1997*, ser. LNCS, vol. 1201. Springer, 1997, pp. 48–62.

[7] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Satisfiability*. IOS Press, 2009, pp. 825–885.

[8] S. Ranise and C. Tinelli, "The SMT-LIB standard: Version 1.2," 2006.

[9] R. Alur and D. L. Dill, "A theory of timed automata," *Theor. Comp. Sci.*, vol. 126, no. 2, pp. 183–235, 1994.

[10] G. Audemard, A. Cimatti, A. Kornilowicz, and R. Sebastiani, "Bounded model checking for timed systems," in *Proc. FORTE 2002*, ser. LNCS, vol. 2529. Springer, 2002, pp. 243–259.

[11] M. Sorea, "Bounded model checking for timed automata," *Elect. Notes Theor. Comp. Sci.*, vol. 68, no. 5, pp. 116–134, 2002.

[12] B. Wozna, A. Zbrzezny, and W. Penczek, "Checking reachability properties for timed automata via SAT," *Fundam. Inform.*, vol. 55, no. 2, pp. 223–241, 2003.

[13] F. Yu, B.-Y. Wang, and Y.-W. Huang, "Bounded model checking for region automata," in *Proc. FORMATS/FTRTFT 2004*, ser. LNCS, vol. 3253. Springer, 2004, pp. 246–262.

[14] M. C. Browne, E. M. Clarke, and O. Grumberg, "Characterizing finite kripke structures in propositional temporal logic," *Theor. Comput. Sci.*, vol. 59, pp. 115–131, 1988.

[15] R. Alur, C. Courcoubetis, and D. L. Dill, "Model-checking in dense real-time," *Inf. Comp.*, vol. 104, no. 1, pp. 2–34, 1993.

[16] B. Dutertre and L. M. de Moura, "A fast linear-arithmetic solver for DPLL(T)," in *Proc. CAV 2006*, ser. LNCS, vol. 4144. Springer, 2006, pp. 81–94.

[17] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Proc. TACAS 1999*, ser. LNCS, vol. 1579. Springer, 1999, pp. 193–207.

[18] A. Biere, K. Heljanko, T. Junttila, T. Latvala, and V. Schuppan, "Linear encodings of bounded LTL model checking," *Logical Methods in Computer Science*, vol. 2, no. 5:5, pp. 1–64, 2006.