Electronic Notes in Theoretical Computer Science 65 No. 6 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume65.html 16 pages

Parking Can Get You There Faster

Model Augmentation to Speed up Real-Time Model Checking [2]

M. Oliver Möller [1]

BRICS, University of Aarhus
Ny Munkegade, building 540
DK - 8000 Århus C, Denmark [3]

Abstract

We present an approximation technique that can render real-time model checking of safety and universal path properties more efficient. It is beneficial, when loops lead to repetition of control situations. Basically we augment a timed automata model with carefully selected extra transitions. This increases the size of the state-space, but potentially decreases the number of symbolic states to be explored by orders of magnitude.

We give a formal definition of a timed automata formalism, enriched with basic data types, hand-shake synchronization, urgency, and committed locations. We prove by means of a trace semantics that if a safety property can be established in the augmented model, it also holds for the original model.

We extend our technique to a richer set of properties that can be decided via a set of traces (universal path properties). In order for universal path properties to carry over to the original model, the semantics of the timed automata formalism is formulated relative to the applied augmentation.

Our technique is particularly useful in systems, where a scheduler dictates repetition of control over elapsing time. As a typical example we mention translations of LEGO® RCX™ programs to Uppaal models, where the Round-Robin scheduler is a static entity. We allow scheduler and associated tasks to "park" until some timing or environmental conditions are met.

We apply our technique on a bricks sorter model for a safety property and report run-time data.

[1] Email: omoeller@brics.dk

[2] A more detailed version of this paper is available as part of the author's thesis, see http://www.brics.dk/~omoeller/papers.html.

[3] Basic Research in Computer Science, funded by the Danish National Research Foundation.

1 Introduction

Failures of safety-critical systems are highly dangerous to human lives, and at minimum incredibly costly. Errors in mass products are hard to correct once the first 100˙000 units have been shipped.

This entails the strong need to validate the *correctness* of a system, or better: to establish that a given design or implementation meets its specification. *Formal Methods* can be seen as a collection of systematic techniques to achieve this. As opposed to simulation or testing, *formal verification* amounts to proving, that a formal model of a system satisfies a set of properties, often expressed by means of a temporal logic. In the *model checking* approach (e.g. [6]) a system is seen as a structure interpreting a formula and potentially satisfying it (in short: $Sys \models \varphi$). Establishing or refuting $Sys \models \varphi$ is then performed fully automatically by specialized model checking algorithms.

Commonly a discrete control graph is used to represent the states of a system, while transitions between states reflect evolution over time. In hybrid models [3] this evolution entails a continuous change of variables according to a mathematical description, usually a differential equation. Even for low degrees of freedom the model checking problem then is undecidable.

Real-time models can be seen as a special version of hybrid models, where the continuous parts are *clocks* that increase their values over time according to a constant slope. As long as all clocks are running at the same speed (slope 1 over time), the model checking problem remains decidable [2]. A popular and well-studied formalism for this are *timed automata* [4]. General decidability results for fixed formalism—both in the language of the model and the logic—were followed by a number of model checking tools, like HyTech [9] for hybrid models, or Kronos [13] and Uppaal [11] for real-time models. Since the state space of the underlying models is fundamentally uncountable, the tools have to resort to symbolic techniques. Here the continuous part is captured by a finite quotient.

For real-time systems, the timed computation tree logic (TCTL, see [2]) is of particular interest, since it can express a large number of naturally occurring specification properties like safety and timed response. Especially timed reachability is the subject of intensive research, since it is powerful enough to capture many routine checks and can be verified more efficiently than general TCTL.

The State-Space Explosion in Real-Time Systems

Model checking techniques suffer from a phenomenon called *state explosion problem*. In dense real-time, this is understood as the excessive number of symbolic states.[4] A number of optimization techniques was developed that

[4] In dense time there exists an intermediate clock evaluation between any two subsequent clock evaluations, thus the state space is infinite. For certain syntactic restrictions, however,

aim to enhance the scope of algorithmic treatment. Examples for such optimizations are reducing the number of clocks wherever possible [8] or reducing memory consumption [12]. Industrial sized examples still present a serious challenge, since expanding their state space often exceeds the memory of the machine or the patience of the user.

A promising approach is to alter a model checking problem $Sys \models \varphi$ into a simpler one, $Sys^\star \models \varphi$, that is conservative in the sense that a validity of the latter implies validity of the former. We construct $Sys^\star$ by *adding* carefully selected parts to the model. This increases the size of the reachable state-space, but can reduce the number of symbolic states that have to be explored. This (seemingly) paradox can best be explained for timed systems that contain small loops in the control structure. If the same control situation occurs repeatedly, the (potentially numerous) corresponding symbolic states may be incomparable, since time has shifted by a delay. Now if a large number of loops with small delays is bypassed by one added step with a large delay, this yields a large set of clock evaluations. If the smaller sets are contained in this (added) larger set, the corresponding steps vanish from the list of symbolic states to be explored.

We use the model checking engine of Uppaal [11] for our experiments, and consequently keep to the Uppaal timed automata model for the description of our technique. However, our technique is general enough to be applied on other timed automata dialects. As a class of problems that is likely to benefit from our technique, we mention Uppaal models of control programs running in LEGO® RCX™ micro-controllers. There exist automatic translations of LEGO® RCX™ programs to Uppaal models [10]; our technique can make direct use of the program as a hint on where to apply augmentations.

Plan.

The paper is organized as follows. Section 2 introduces the Uppaal timed automata model with formal syntax and semantics. Section 3 defines our model augmentation technique and proves it sound for safety properties. In Section 4 we extend our technique to universal path properties. In Section 5 we apply model augmentation on a bricks sorter. We conclude with a summary.

2 Uppaal Timed Automata

We give formal syntax of the timed automata model as used in Uppaal [11,5]. Basically this means networks of timed automata with discrete data and handshake synchronization. The formal semantics is defined by associating models with sets of traces.

Uppaal is a tool for modeling, simulation and verification of real-time systems, developed jointly by BRICS at Aalborg University and the Department of Computer Systems at Uppsala University. Typical application areas

the number of sets of clock evaluations that can be treated uniformly is always finite [2].

include real-time controllers and communication protocols, in particular those where timing aspects are critical. More information and documentation can be found on the home page. [5]

2.1 Formal Syntax of Uppaal *Timed Automata*

We define the formal syntax of Uppaal models as a parallel composition of processes, where in each process control locations are connected by transitions and equipped with various labels.

As a preliminary, we assume disjoint sets of *variables* (*Vars*), *clocks* (*Cl*), and *synchronization channels* (*Ch*). Variables denote either integers or bounded arrays of integers. Expressions are constructed over variables, clocks, and integer constants. *Assignments* are of the form `v := expr`, and either variable assignments or clock resets. In the first case, `v` is an integer variable (or a position in an array) and `expr` is an arithmetic expression over variables and constants. In the second case, `v` is a clock and `expr` is required to be the constant `0`. A *guard* is a conjunction of Boolean expressions over variables and clock constraints of form `x`$\sim$`const` or `x-y`$\sim$`const`, where `x,y` are clocks, $\sim \in \{$`<`, `<=`, `==`, `>=`, `>`$\}$, and `const` is an integer constant. An *invariant* is of the form `x`$\sim$`const`, where `x,y` are clocks, $\sim \in \{$`<`, `<=`$\}$, and `const` is an integer constant. A *synchronization* is of the form `s!` or `s?`, where `s` is a synchronization channel. We understand synchronization to be hand-shake, i.e., one transition equipped with `s!` and one equipped with `s?` can be taken simultaneously.

For simplicity, we assume a set of labels—*Labels*—that ranges over syntactically correct invariants, assignments, guards, and synchronizations. As a well-formedness condition, labels of the described syntax are constrained to occur only in the appropriate places, contain only declared variables, clocks, and synchronization channels, and arrays are used in a syntactically correct way.

Locations can optionally be declared *urgent* or *committed*. In both cases the location has to be left before time elapses. Leaving committed locations has precedence over taking other transitions. If synchronization channels are declared urgent, transitions synchronizing on these channels have precedence over time delay.

Def. 1 (Uppaal process)
An Uppaal process A *is a tuple* $\langle L, T, Type, l^0 \rangle$*, where*

- L *is a set of locations,*
- T *is a set of transitions* $l \xrightarrow{g,s,\boldsymbol{a}} l'$*, where* $l, l' \in L$*,* g *is a guard,* s *is a synchronization label (optional), and* $\boldsymbol{a}$ *is a list of assignments (possibly empty); we call* l *the* source *and* l' *the* target *of the transition,*
- $Type : L \to \{o, u, c\}$ *is the type function for locations (ordinary, urgent, or committed),*

[5] `http://www.uppaal.com`

- $l^0 \in L$ *is the initial location.*

We use the following access functions to refer to invariants, guards, synchronizations, and assignments.

- $\mathit{Inv} : L \to \mathit{Labels}$ maps to the invariant of a location (possibly **true**),
- $\mathit{Guard} : T \to \mathit{Labels}$ maps to the guard of a transition (possibly **true**),
- $\mathit{Sync} : T \to \mathit{Labels} \cup \{\varnothing\}$ maps to the synchronization label of a transition (if any), and
- $\mathit{Assign} : T \to \mathit{Labels}^*$ maps to the assignments associated with a transition (possibly the empty list).

Def. 2 (Uppaal model)
An Uppaal model *is a tuple* $\langle \boldsymbol{A}, \mathit{Vars}, \mathit{Cl}, \mathit{Ch}, \mathit{Type}\rangle$*, where*

- $\boldsymbol{A}$ *is a vector of processes* $A_1, \ldots, A_n$*;*
  we use the index i *to refer to* A_i*-specific parts* L_i*,* T_i*,* Type_i*, and* l_i^0*,*
- Vars *is a set of variables, corresponding to bounded integers and arrays,*
- Cl *is a set of clocks,* $\mathit{Cl} \cap \mathit{Vars} = \varnothing$*,*
- Ch *is a set of synchronization channels,* $\mathit{Ch} \cap \mathit{Vars} = \varnothing$ *and* $\mathit{Ch} \cap \mathit{Cl} = \varnothing$*,*
- Type *is a polymorphic type function extending the* Type_i*, i.e.,* Type *maps*
  - *locations to* $\{\mathsf{o}, \mathsf{u}, \mathsf{c}\}$ *(ordinary, urgent, or committed—according to* Type_i*),*
  - *channels to* $\{\mathsf{o}, \mathsf{u}\}$ *(ordinary or urgent),*
  - *variables to* $\{\mathsf{int}, \mathsf{array}\}$*.*

  We use o*,* u*,* c*,* int*, and* array *as predicates, i.e., for a synchronization channel* s *the expression* $\mathsf{u}(s)$ *evaluates to* **true***, if and only if* $\mathit{Type}(s) = \mathsf{u}$*.*

Def. 3 (configuration) *A* configuration *of an* Uppaal *model* $\langle \boldsymbol{A}, \mathit{Vars}, \mathit{Cl}, \mathit{Ch}, \mathit{Type}\rangle$ *is a triple* $(\boldsymbol{l}, \mathrm{e}, \nu)$*, where* $\boldsymbol{l}$ *is the control vector,* e *is the environment for discrete variables, and* ν *is the clock evaluation, i.e.:*

- $\boldsymbol{l} = (l_1, \ldots, l_n)$*, where* $l_i \in L_i$ *is a location of process* A_i*,*
- $e : \mathit{Vars} \to (\mathbb{Z})^*$ *maps every variable* v *either to a value (if* $\mathsf{int}(v)$*) or to a tuple of values (in case of* $\mathsf{array}(v)$*).*
- $\nu : \mathit{Cl} \to \mathbb{R}_{\geq 0}$ *maps every clock to a non-negative real number. For* $d > 0$*, the notation* $(\nu + d) : \mathit{Cl} \to \mathbb{R}_{\geq 0}$ *describes the function "*ν *shifted by* d*" in the following sense:* $\forall x \in \mathit{Cl}.\ (\nu(x) + d) = \nu(x) + d$*.*

For simplicity we allow only one initial configuration, denoted $\big((l_1^0, \ldots, l_n^0), [\mathit{Vars} \mapsto (0)^*], [\mathit{Cl} \mapsto 0]\big)$, where all processes are in their initial location, all variables and all array positions evaluate to 0, and all clocks are 0.

A *local property* is a Boolean expression φ over Vars, Cl, and terms $A_i.l_i$. In any configuration of a model φ is either **true** or **false**. We use the notation $e, \nu \models \varphi$ to state that a local property φ holds true under the evaluations e, ν for the contained variables and clocks. Analogously we write $(\boldsymbol{l}, e, \nu) \models \varphi$ in the case that φ contains expressions of the form $A_i.l_i$. $A_i.l_i$ is **true** if and only if process A_i is in location l_i, i.e., $\boldsymbol{l} = (\ldots, l_i, \ldots)$.

2.2 Trace Semantics of Uppaal *Timed Automata*

We associate an Uppaal model M with a—typically uncountable—set $\mathcal{T}(M)$ of traces that are either infinite or maximally extended (deadlocked). The model M does then satisfy the safety property ψ, if no trace in $\mathcal{T}(M)$ leads to a configuration that violates ψ. An universal path property is true in M, if all traces conform to this property.

We start by formulating simple action, synchronized action, and delay steps. To modify the control vector $\boldsymbol{l}$, we use the notation $\boldsymbol{l}[l'_i/l_i]$ to indicate, that at position i, l_i is replaced by l'_i, and the other positions do not change. We readily use assignments $\boldsymbol{a}$ as transformers on the function e (and ν) and write $\boldsymbol{a}(e)$ (and $\boldsymbol{a}(\nu)$) for the resulting evaluations. The assignments in $\boldsymbol{a}$ are considered to be applied in the order of occurrence.

Def. 4 (simple action step) *For a configuration* $(\boldsymbol{l}, e, \nu)$*, a* simple action step *is enabled, if there is a transition* $l_i \xrightarrow{g,\boldsymbol{a}} l'_i \in T_i$*,* l_i *in* $\boldsymbol{l}$*, such that*

(i) $e, \nu \models g$*, and*

(ii) $\boldsymbol{a}(e), \boldsymbol{a}(\nu) \models \mathit{Inv}(l'_i)$*, and*

(iii) *if* $\exists l_c$ *in* $\boldsymbol{l}$ *with* $c(l_c)$*, then* $c(l_i)$*.*

We abbreviate this with $(\boldsymbol{l}, e, \nu) \xRightarrow{\boldsymbol{a}} (\boldsymbol{l}[l'_i/l_i], \boldsymbol{a}(e), \boldsymbol{a}(\nu))$*.*

Def. 5 (synchronized action step) *For a configuration* $(\boldsymbol{l}, e, \nu)$*, a* synchronized action step *is enabled if and only if for a channel* s *there exist two transitions* $l_i \xrightarrow{g_i, s!, \boldsymbol{a}_i} l'_i \in T$ *and* $l_j \xrightarrow{g_j, s?, \boldsymbol{a}_j} l'_j \in T$*,* l_i, l_j *in* $\boldsymbol{l}$*,* $i \neq j$*, such that*

(i) $e, \nu \models g_i \wedge g_j$*, and*

(ii) $\boldsymbol{a}_j(\boldsymbol{a}_i(e)), \boldsymbol{a}_j(\boldsymbol{a}_i(\nu)) \models \mathit{Inv}(l'_i) \wedge \mathit{Inv}(l'_j)$*, and*

(iii) *if* $\exists l_c$ *in* $\boldsymbol{l}$ *with* $c(l_c)$*, then* $c(l_i) \vee c(l_j)$*.*

We abbreviate this with $(\boldsymbol{l}, e, \nu) \xRightarrow{s!?} \big(\boldsymbol{l}[l'_i/l_i][l'_j/l_j],\ \boldsymbol{a}_j(\boldsymbol{a}_i(e)),\ \boldsymbol{a}_j(\boldsymbol{a}_i(\nu))\big)$*.*

Def. 6 (delay step) *For a configuration* $(\boldsymbol{l}, e, \nu)$*, a* delay step *with delay* d *is enabled, if and only if all of the following holds.*

(i) $e, (\nu + d) \models \bigwedge_i \mathit{Inv}(l_i)$*,*

(ii) $\forall l_i$ *in* $\boldsymbol{l}$*.* $\neg c(l_i)$*,*

(iii) *no action step leaving an urgent location is enabled, i.e., if* $\mathrm{u}(l_i)$ *for some* $l_i \in \boldsymbol{l}$*, then* $\neg\big((\boldsymbol{l}, e, \nu) \xRightarrow{\boldsymbol{a}} (\boldsymbol{l}', e', \nu')\big) \ \wedge\ \neg\big((\boldsymbol{l}, e, \nu) \xRightarrow{s!?} (\boldsymbol{l}', e', \nu')\big)$*, and*

(iv) *no synchronized action on a urgent channel is enabled, i.e.,*
$(\boldsymbol{l}, e, \nu) \xRightarrow{s!?} (\boldsymbol{l}', e', \nu')$ *implies* $\neg\mathrm{u}(s)$*.*

We abbreviate this with $(\boldsymbol{l}, e, \nu) \xRightarrow{d} (\boldsymbol{l}, e, (\nu + d))$*.*

Def. 7 (timed trace)
A trace *of a* Uppaal *model* $\langle \boldsymbol{A}, \mathit{Vars}, \mathit{Cl}, \mathit{Ch}, \mathit{Type}\rangle$ *is a finite or infinite sequence of configurations* $\{(\boldsymbol{l}, e, \nu)\}^K = ((\boldsymbol{l}, e, \nu)^0, (\boldsymbol{l}, e, \nu)^1, ..)$ *with* K *steps,* $K \in \mathbb{N} \cup \{\infty\}$*, such that*

(i) $(\boldsymbol{l}, \mathrm{e}, \nu)^0 = \big((l_1^0, \ldots, l_n^0), [Vars \mapsto (0)^*], [Cl \mapsto 0]\big)$,

(ii) *for every* $k < K$, *the two subsequent configurations* k *and* $k+1$ *are connected via a simple action step, a synchronized action step, or a delay step, i.e.,*

$$(\boldsymbol{l}, \mathrm{e}, \nu)^k \stackrel{\boldsymbol{a}}{\Longrightarrow} (\boldsymbol{l}, \mathrm{e}, \nu)^{k+1} \quad \textit{or}$$

$$(\boldsymbol{l}, \mathrm{e}, \nu)^k \stackrel{s!?}{\Longrightarrow} (\boldsymbol{l}, \mathrm{e}, \nu)^{k+1} \quad \textit{or}$$

$$(\boldsymbol{l}, \mathrm{e}, \nu)^k \stackrel{d}{\Longrightarrow} (\boldsymbol{l}, \mathrm{e}, \nu)^{k+1} \quad \textit{, and}$$

(iii) *traces are infinite or maximally extended, i.e., if* $K < \infty$, *then for* $(\boldsymbol{l}, \mathrm{e}, \nu)^K$ *no action step or delay step is enabled.*

Def. 8 (trace semantics) *Let* M *be a* Uppaal *timed automata model. Then the* trace semantics *of* M, *written* $\mathcal{T}(M)$, *is the set of traces that can be constructed according to Definition 7.*

We are now ready to formally define timed reachability and timed safety.

Def. 9 (timed reachability/safety) *We define* timed reachability *as a binary relation between an* Uppaal *timed automata model* M *and reachability predicates* `E<>` φ, *where* φ *is a local property for* M.

$$M \models \texttt{E<>}\,\varphi \quad \textit{if an only if} \quad \exists\{(\boldsymbol{l}, \mathrm{e}, \nu)\}^K \in \mathcal{T}(M).\ \exists k < K.\ (\boldsymbol{l}, \mathrm{e}, \nu)^k \models \varphi.$$

Timed safety, denoted by $M \models$ `A[]` φ, *is then dual to timed reachability:*

$$M \models \texttt{A[]}\ \varphi \quad \textit{if and only if} \quad \neg\,(\,M \models \texttt{E<>}\ \neg\varphi\,).$$

3 Model Augmentation

We introduce *model augmentation* as a technique to enrich the behavior of a given model. This is shown to be conservative with respect to timed safety properties in one direction. Model augmentation entails an increase in the size of the reachable state space. Nevertheless, the number of symbolic states to explore can be smaller, because larger portions can be covered in one step. We illustrate this phenomenon by a simple delay-loop example.

We define the transformation of an original Uppaal model formally as follows.

Def. 10 (model augmentation) *Let* $M = \langle \boldsymbol{A}, Vars, Cl, Ch, Type\rangle$ *be an* Uppaal *model, and let* $\mathfrak{A} = \langle l_i \xrightarrow{g,\boldsymbol{a}} l', L_{\mathfrak{A}}, T_{\mathfrak{A}}, Type_{\mathfrak{A}}\rangle$, *where*

- $l_i \in L_i$ *for some process* A_i, *that is part of* $\boldsymbol{A}$, *where we require* $o(l_i)$; *we call* l_i *the* augmentation point *of* $\mathfrak{A}$,
- g *a guard,*
- $\boldsymbol{a}$ *a list of assignments,*
- $l' \in L_{\mathfrak{A}}$, $L_{\mathfrak{A}}$ *a set of fresh locations,* $L_{\mathfrak{A}} \cap (\bigcup L_i) = \varnothing$,
- $T_{\mathfrak{A}}$ *a set of transitions, such that all sources are in* $L_{\mathfrak{A}}$ *and all targets are in* $L_{\mathfrak{A}} \cup \bigcup L_i$, *and*
- $Type_{\mathfrak{A}} : L_{\mathfrak{A}} \to \{\mathrm{o}, \mathrm{u}, \mathrm{c}\}$ *the type function for the fresh locations.*

	M			$Aug_{\mathfrak{A}}(M)$		
LARGE	#states	time[sec]	memory[KB]	#states	time[sec]	memory[KB]
10	8	0.01	376	9	0.01	448
100	35	0.01	440	9	0.01	376
1000	305	0.04	424	9	0.01	440
10˙000	3˙005	1.51	1˙704	9	0.01	440
100˙000	30˙005	175.21	5˙440	9	0.02	416
1˙000˙000	300˙005	22˙449.94	42˙792	9	0.02	400

Table 1

Then $Aug_{\mathfrak{A}}(M)$ *is the* Uppaal *model* $\langle \boldsymbol{A'}, Vars, Cl, Ch, Type' \rangle$ *that enriches* M *in the following sense:*

(i) $\boldsymbol{A'} = A_1, \ldots, A_{i-1}, A'_i, A_{i+1}, \ldots, A_n$, *with*
$A'_i = \langle L_i \uplus L_{\mathfrak{A}}, T_i \uplus T_{\mathfrak{A}} \uplus \{l_i \xrightarrow{g,\boldsymbol{a}} l'\}, Type_i \uplus Type_{\mathfrak{A}}, l_i^0 \rangle$, [6]

(ii) $Type'$ *extends* $Type$ *by mapping locations* $l_{\mathfrak{A}} \in L_{\mathfrak{A}}$ *to* $Type_{\mathfrak{A}}(l_{\mathfrak{A}})$.

We call $\mathfrak{A}$ *the* model augmentation *and* $Aug_{\mathfrak{A}}(M)$ *the* augmented model.

Example 11 (delay loop) Consider a model M with a single process P (Figure 1). P performs a number of delay loops of duration `10`, and leaves the loop when a total delay `LARGE` was reached. When the property `E<> P.QUICK` is verified in forward state space exploration, a large number of these delay loops are explored before it is established that the location `QUICK` indeed cannot be reached. Figure 1 shows the augmented model $Aug_{\mathfrak{A}}(M)$ on the right, where

$$\mathfrak{A} = \left\langle T \xrightarrow{\texttt{x <= LARGE}} \texttt{AUGMENT}, \{\texttt{AUGMENT}[\mathit{Inv}\text{:}\ \texttt{x} <= \texttt{LARGE}]\}, \ \{\texttt{AUGMENT} \to \texttt{S}\} \right\rangle.$$

The augmented process can be understood to "park" in location `AUGMENT`, until time has progressed enough to pass the guard $x >$ `LARGE`.

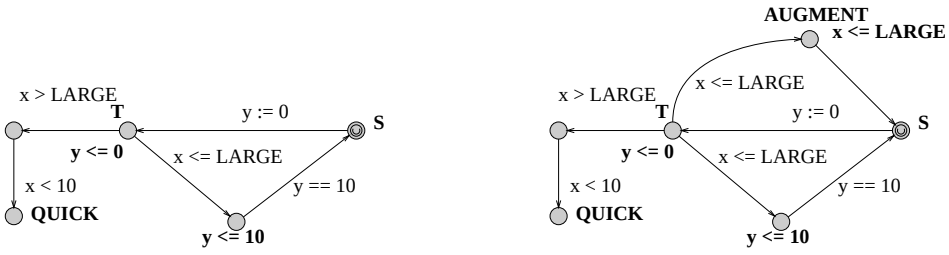


Fig. 1. The original process P (left), and P with model augmentation $\mathfrak{A}$ (right).

Table 1 shows data for forward reachability analysis with Uppaal. [7] The

[6] The symbol $\uplus$ denotes disjoint union.

[7] All run-time measurements were performed with the command-line version of Uppaal, `verifyta` 3.1.58, executing on a UltraSPARC-II 300 MHz under SolarisOS.

number of explored states in the original model with process P (left) increases linearly in the parameter `LARGE`, whereas this numbers stays constant, when $Aug_{\mathfrak{A}}(P)$ (right) is used.

3.1 Soundness of Model Augmentation

Our technique is sound for proving that some local property holds invariantly.

Theorem 12 (soundness) *Let $M = \langle \boldsymbol{A}, Vars, Cl, Ch, Type \rangle$ be an* Uppaal *model and φ be a local property. Then for any model augmentation $\mathfrak{A} = \langle l_{\mathfrak{A}} \xrightarrow{g,\boldsymbol{a}} l'_{\mathfrak{A}}, L_{\mathfrak{A}}, T_{\mathfrak{A}} \rangle$ with $l_{\mathfrak{A}} \in L_i$ for some i, the following holds.*

$$M \models \texttt{E<>}\ \varphi \qquad \Rightarrow \qquad Aug_{\mathfrak{A}}(M) \models \texttt{E<>}\ \varphi$$

Proof. We show that $\mathcal{T}(M) \subseteq \mathcal{T}(Aug_{\mathfrak{A}}(M))$. For this it suffices to show, that for any configuration $\mathbf{s} = (\boldsymbol{l}, e, \nu)$ is reached in M, every enabled step is also enabled in the corresponding configuration $\mathbf{s}_{\mathfrak{A}} = (\boldsymbol{l}, e, \nu)$ in $Aug_{\mathfrak{A}}(M)$.

Assume a simple or synchronized action step is enabled in $\mathbf{s}$. For every transition of M, there is an equivalent transition in $Aug_{\mathfrak{A}}(M)$. By Definition 10, $o(l_{\mathfrak{A}})$, thus $\neg c(l_{\mathfrak{A}})$ and the transition $l_{\mathfrak{A}} \xrightarrow{g,\boldsymbol{a}} l'_{\mathfrak{A}}$ has no precedence over other action steps. Since $\boldsymbol{l}$, e, and ν of $\mathbf{s}$ and $\mathbf{s}_{\mathfrak{A}}$ are identical, the same simple or synchronized action step is then enabled in $\mathbf{s}_{\mathfrak{A}}$.

Assume a delay step of duration d is enabled in $\mathbf{s}$. Then conditions (i) to (iv) from Definition 6 are met. For $\mathbf{s}_{\mathfrak{A}}$, the conditions (i) and (ii) then also hold true, since $\boldsymbol{l}$, e, and ν are identical for $\mathbf{s}$ and $\mathbf{s}_{\mathfrak{A}}$. Condition (iii) is met, since no action step leaving an urgent location is enabled in $\mathbf{s}$. $o(l_{\mathfrak{A}})$ entails $\neg u(l_{\mathfrak{A}})$, thus $l_{\mathfrak{A}} \xrightarrow{g,\boldsymbol{a}} l'_{\mathfrak{A}}$ does not introduce an additional one for $\mathbf{s}_{\mathfrak{A}}$.

As for condition (iv), $l_{\mathfrak{A}} \xrightarrow{g,\boldsymbol{a}} l'_{\mathfrak{A}}$ does not carry a synchronization by definition. Thus no synchronization on an urgent channel can be enabled in $\mathbf{s}_{\mathfrak{A}}$, unless it was also enabled in $\mathbf{s}$, which is not the case.

Thus a delay step of duration d is also enabled in configuration $\mathbf{s}_{\mathfrak{A}}$, completing the proof. □

Corollary 13 (conservative for safety) *Let M be an* Uppaal *model and φ a local property. For any model augmentation $\mathfrak{A}$:*

$$Aug_{\mathfrak{A}}(M) \models \texttt{A[]}\ \varphi \qquad \Rightarrow \qquad M \models \texttt{A[]}\ \varphi$$

3.2 Suitable Augmentations

Though not formally required, model augmentations have to return to the original control structure. Otherwise they never yield an improvement.

Model augmentation adds both to the state space and to the level of nondeterminism. In general this is a bad thing. The modification is only beneficial, if the additional loop cuts out long and tedious repetitions of control sequences that are only distinguishable by the passage of time. I.e., repetitions

modulo a certain clock shift must exist. It is necessarily to apply augmentation in all processes of the model before this phenomenon can be exploited.

It is crucial that the newly introduced loop is taken early in the state space exploration. In forward reachability analysis this can be achieved by using a breadth-first search order. Then one augmented loop is explored before the concrete control returns to the augmentation point. A more rigorous possibility is to modify the model checking algorithm in such a way that the transitions starting model augmentations are explored first.

The challenges for successful model augmentation are

(i) To find promising augmentation points,
(ii) To identify suitable delays, and
(iii) To construct conditions that should trigger a return to the original control structure.

In section 5 we exemplify this on a medium sized example with two parallel tasks, where a Round-Robin scheduler dictates repetitions over time.

4 Model Augmentation for Universal Path Properties

Universal path properties are the fragment of TCTL, where a property can be refuted by a single counter-example trace. We extend our model augmentation technique to be conservative with respect to this richer set of properties. In order to preserve deadlocks, we modify the transition relation relative to the model augmentation.

Universal path properties ζ are of the form

$$\zeta ::= \texttt{A[]}\zeta \;\mid\; \texttt{A<>}\zeta \;\mid\; \zeta \vee \zeta \;\mid\; \zeta \wedge \zeta \;\mid\; \varphi$$

where φ a local property. In particular the definition of unbounded response, $\texttt{A[]}\varphi \Rightarrow \texttt{A<>}\psi$, is equivalent to $\texttt{A[]}\neg\varphi \vee \texttt{A<>}\psi$ and thus is a universal path formula.

The operator `A<>` expresses *inevitability*: at some point in the future, some property will necessarily hold. `A<>` ζ is violated, if there exists either an infinite trace not containing a configuration, where ζ holds, or some finite trace, that reaches a deadlock without passing through a state where ζ holds.

A trace $\sigma = (\mathbf{s}_0, \mathbf{s}_1, \ldots) \in \mathcal{T}(M)$ satisfies an universal path formula ζ at position i according to the following rules:

$$\begin{array}{lcll}
(\sigma, i) & \models & \texttt{A[]}\zeta & \iff \forall j \geq i.(\sigma, j) \models \zeta \\
(\sigma, i) & \models & \texttt{A<>}\zeta & \iff \exists j \geq i.(\sigma, j) \models \zeta \\
(\sigma, i) & \models & \zeta_1 \vee \zeta_2 & \iff (\sigma, i) \models \zeta_1 \text{ or } (\sigma, i) \models \zeta_2 \\
(\sigma, i) & \models & \zeta_1 \wedge \zeta_2 & \iff (\sigma, i) \models \zeta_1 \text{ and } (\sigma, i) \models \zeta_2 \\
(\sigma, i) & \models & \varphi & \iff \mathbf{s}_i \models \varphi
\end{array}$$

A model M satisfies a formula ζ, if for all traces $\sigma = (\mathbf{s}_0, \mathbf{s}_1, \ldots) \in \mathcal{T}(M)$,

$(\sigma, 0) \models \zeta$.[8]

Applying model augmentation with universal path properties raises a technical problem. It could be the case, that the new transition at the augmentation point allows to escape from a deadlock situation, where no further action transitions are possible. In this situation, `A<>` properties in the augmented model could hold, though they do not for the original system.

The solution is conceptually simple; we require, that in the augmentation point, the added transition can only be taken, if another action transition can be taken as well. We formalize this as follows.

Def. 14 (augmented path semantics) *Let M be an* Uppaal *timed automata model and $\mathfrak{A} = \langle l_i \xrightarrow{g,\boldsymbol{a}} l', L_{\mathfrak{A}}, T_{\mathfrak{A}}, Type_{\mathfrak{A}} \rangle$ a model augmentation of M. We define the set of* weak traces *of $Aug_{\mathfrak{A}}(M)$ as the subset of the set of traces according to Definition 7, that is generated from $\mathcal{T}(Aug_{\mathfrak{A}}(M))$ by the following side condition:*

in a configuration $(\boldsymbol{l}, \mathrm{e}, \nu)$ with $l_i \in \boldsymbol{l}$, the action transition $l_i \xrightarrow{g,\boldsymbol{a}} l'$ is only enabled, if another action transition is enabled.

All traces violating this condition are removed from $\mathcal{T}(Aug_{\mathfrak{A}}(M))$ to yield $\mathcal{T}^{\mathfrak{A}}(Aug_{\mathfrak{A}}(M))$. We write $Aug_{\mathfrak{A}}(M) \models^{\mathfrak{A}} \zeta$, if and only if for all traces $\sigma = (\mathbf{s}_0, \mathbf{s}_1, \ldots) \in \mathcal{T}^{\mathfrak{A}}(Aug_{\mathfrak{A}}(M))$, $(\sigma, 0) \models \zeta$.

Proposition 15 (conservative over universal path properties) *Let M be an* Uppaal *timed automaton model and ζ a universal path property. For any model augmentation $\mathfrak{A}$:*

$$Aug_{\mathfrak{A}}(M) \models^{\mathfrak{A}} \zeta \qquad \Rightarrow \qquad M \models \zeta$$

Example 16 *For the* Uppaal *model M with the single process P in Figure 1:*

$Aug_{\mathfrak{A}}(M) \models^{\mathfrak{A}}$ `A[]`$(\neg P.T \vee$ `A<>` $P.S)$, and thus $M \models$ `A[]`$(\neg P.T \vee$ `A<>` $P.S)$. Note that whenever transition $T \rightarrow$ `AUGMENT` is enabled, then due to the invariant $y \leq 0$ in T, one of the original transitions in M is enabled as well. Thus, the side-condition is always fulfilled for $T \rightarrow$ `AUGMENT`, and $\mathcal{T}(Aug_{\mathfrak{A}}(M)) = \mathcal{T}^{\mathfrak{A}}(Aug_{\mathfrak{A}}(M))$.

5 Faster Verification of a Bricks Sorter

We demonstrate how to apply our model augmentation technique on a special class of examples, namely Uppaal models of task-based LEGO® RCX™ programs, which can be automatically translated to Uppaal models. We use the bricks sorter example from [10] as a case study.

The bricks sorter model is augmented in the places where control loops in

[8] Since our semantics of a model contains traces where time converges, `A<>` properties hold true only under the *time progress assumption*: a global reference clock must eventually exceed any time bound, compare [1, 2].

the structure were detected. For safety properties this yields a speed-up in terms of model checking time.

5.1 The Bricks Sorter Model

The bricks sorter (Figure 2) is a machine consisting of a conveyor belt, a light sensor, and a kick-off arm. Red and black bricks are transported on the conveyor belt past the sensor, which is sensitive enough to distinguish the two colors. Some time later, the kick-off arm can push a brick off the belt. A controller coordinates sensor and kick-off arm and tries to make sure that every black brick is pushed off, while every red brick is allowed to pass.

In a physical implementation, this system was built in LEGO®, with a RCX™ Mindstorm micro-controller as the control unit. This controller executes up two ten tasks that are organized by a deterministic scheduler in Round-Robin fashion. Two tasks `main` and `kick-off` are used in the RCX™ program, which are translated to three Uppaal processes [10]. Three processes model the environment. and one `Hurry_Dummy` was added to permanently offers synchronization on the urgent channel `Hurry`. This composes the Uppaal model *Sorter*. We want to establish a safety property stating that the second brick is kicked off, regardless of when precisely it enters the belt. Formally we model check *Sorter* $\models$ `A[]`$\neg$`black_brick2.PASSED`.

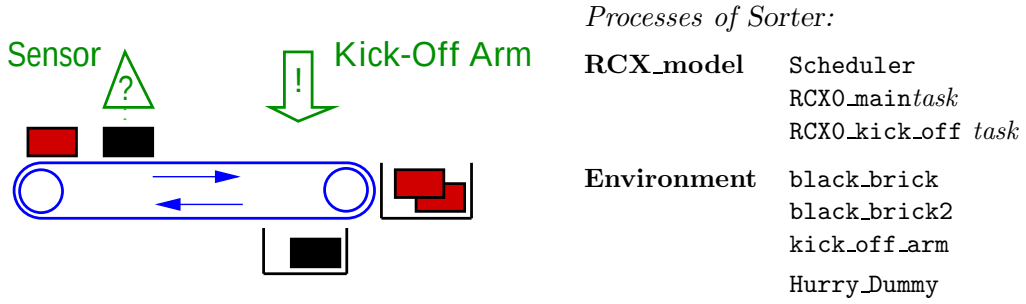


Fig. 2. Schematic description of the bricks sorter.

5.2 Augmentation of the Bricks Sorter Model

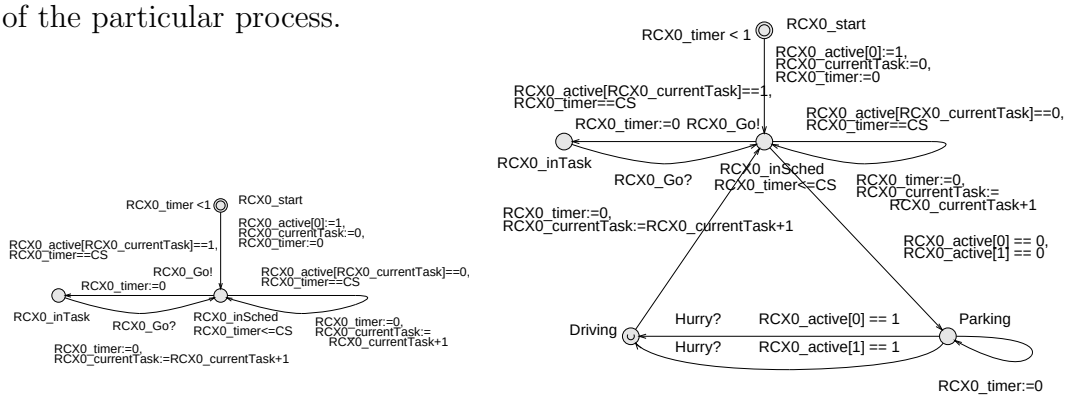
However, some classes of examples exhibit a fairly regular structure, where this technique can be applied more systematically. We mention here LEGO® RCX™ micro-controller, where NQC programs can be automatically translated to Uppaal models for analysis. We use the translation of NQC programs to Uppaal models from [10]. Original tasks are translated to Uppaal processes and `WAIT-UNTIL` statements correspond to augmentation points. Additionally, the Scheduler has to be modified to allow a parking state.

One problem with model checking the Uppaal model *Sorter* is that at some points in time, the processes perform a busy loop. More precisely, when the sensor waits to get sight of a brick or the kick-off arm waits for a brick

to be in the right position, all processes return to the same control situation (after some delay) very often.

We try to avoid this busy loop by an appropriate augmentation of the model. This can be understood as allowing processes to *park* in some situations, until an enabling timing condition holds true.

Starting with the Uppaal model *Sorter* we define a sequence of model augmentations. The processes `Scheduler`, `RCX0_main`, and `RCX0_kick_off` are augmented. The augmentation points are chosen according to the nature of the particular process.



(i) Original `Scheduler` Process *(ii)* Augmented Scheduler Process

Fig. 3. The Round-Robin Scheduler repeatedly toggles through the list of tasks.

```
*** Task 0 = main
...
031 InType      2, Switch
034 InMode      2, Boolean
037 OutDir      A, Fwd
039 OutMode     A, On
041 OutPwr      A, 1
045 OutDir      B, Fwd
047 OutMode     B, On
049 OutPwr      B, 6
053 Display     1
057 StartTask   1
059 Test        Input(0) <= var[4], 70
067 Jump        59
070 ...
```

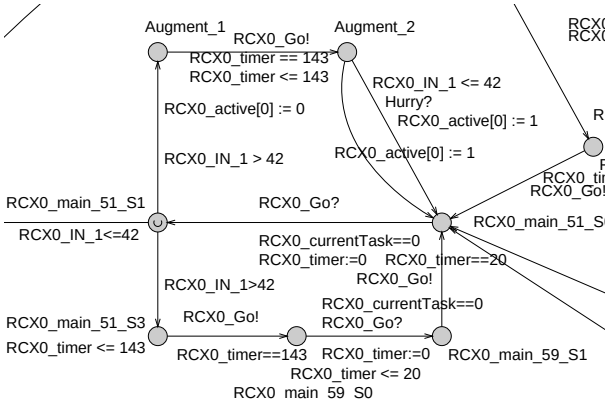


Fig. 4. Part of a LEGO® RCX™ task and the corresponding Uppaal process.

Scheduler.

Figure 3 *(i)* displays the Uppaal process `Scheduler`. It uses the array `RCX0_active` and the integer variable `RCX0_current_task` to keep track of the next task to release. It does so by taking the transition to `RCX0_inTask`, if possible, and otherwise idles via the self loop at `RCX0_inSched`.

If one respective task is active, execution of this task is released (`RCX0_inTask`). The task executes an instruction, and then hands back control to the scheduler by synchronizing on channel `RCX0_go`. If the respective task is inactive, it is skipped (self-loop to the right).

	#explored states	average number of successors	#deadlocks	time [sec]	memory [KB]
Sorter	151˙103	1.28	0	86.84	1˙840
$Aug^*_{\mathfrak{A}}(Sorter)$	22˙966	2.09	20	21.15	2˙512

Table 2

Hereafter, the scheduler moves on to the next task. The variable `RCX0_current_task` wraps around to 0, when the number of existing processes is exceeded.

The augmented scheduler *(ii)* is shown on the right. If all tasks are inactive, the location `Parking` can be reached. If one of the tasks become active again, this location has to be left immediately. This is achieved via synchronization on urgent channel `Hurry` and declaring the location `Driving` urgent.

Tasks.

In the processes `RCX0_main` and `RCX0_kick_off` the conditional tests cause loops in the control structure. Model augmentations are applied in eight places. Six of them were wait conditions for conditions to hold true, like the one shown in Figure 4. The remaining two are allowing optional time delay, whenever progress depends on timing conditions to be met.

This amounts to nine model augmentations, that add 16 locations and 34 transitions in total. We refer to the obtained model as $Aug^*_{\mathfrak{A}}(Sorter)$. The comparison of *Sorter* and $Aug^*_{\mathfrak{A}}(Sorter)$ is given in Table 2. All runs use the optimization options `-sWabA -S 2` (active clock reduction, breadth-first search, convex hull approximation, minimal space. On $Aug^*_{\mathfrak{A}}(Sorter)$, forward state space exploration runs four times faster, but consumes slightly more memory. This is a consequence the applied convex-hull optimization. In *Sorter* the convex hull is constructed over the many encountered zones, each one shifted by small delay. I.e., at most one zone is stored for each discrete part. $Aug^*_{\mathfrak{A}}(Sorter)$ additionally needs to store symbolic states for the new control locations.

$Aug^*_{\mathfrak{A}}(Sorter)$ exhibits a considerably higher average number of successors, which can be understood as additional non-determinism. Also, $Aug^*_{\mathfrak{A}}(Sorter)$ yields additional deadlock states, apparently due to unfortunate timing clashes. In general this is undesirable, for it remains unclear whether the original model is deadlock-free. In this class of examples, however, the original model is *by construction* deadlock-free and thus the deadlocks are necessarily spurious.

Note that $Aug^*_{\mathfrak{A}}(Sorter)$ is also good for proving other safety properties. If they hold in $Aug^*_{\mathfrak{A}}(Sorter)$, the whole state-space is expanded symbolically in one run; thus the run-time data would be identical (modulo small run-time differences in evaluating the invariant φ for one symbolic configuration).

In the model checking run, we make use of the convex-hull approximation technique. This approximation increases the number of reachable states but is

conservative with safety properties in one direction. Without this option, the large number of created symbolic states in this example exceeded the available 1 GB of memory, both for *Sorter* and for $Aug^*_{\mathfrak{A}}(Sorter)$.

6 Summary

Model augmentation is a specialized approximation technique that is safe for various timed automata dialects. Note that we restrain from the *pruning* of processes, i.e., existing behavior of the system is never prohibited. This allows for a general soundness proof of our technique and makes combination with other approximation techniques possible.

In our application on LEGO® RCX™ programs, the savings are not drastic. Nevertheless the run-time improvements in a brick-sorter example demonstrates that the technique has potential. The time savings were roughly 75%, but slightly more memory was consumed with the augmented model. More experiments are needed to determine whether this is specific only to this example.

The augmentation points of the tasks can be either derived from the control structure of the Uppaal model, or even directly from the LEGO® RCX™ program. There exists a translation translation from these programs to corresponding Uppaal models [10]. It is possible to modify this translation to directly compute an augmented version of the Uppaal models, providing full automation for this optimization technique in this class of application.

Our technique is related to the convex hull over-approximation in [7]. However, it has a finer granularity in the choice of the approximation and can even be combined with convex hull. Other than convex-hull, our technique prescribes a way to modify a model checking algorithm such that universal path properties carry over from the augmented model to the original one.

Acknowledgments. We thank Kim G. Larson for many helpful comments.

References

[1] Martin Abadi and Leslie Lamport. An Old-Fashioned Recipe for Real Time. In *Proc. of REX Workshop "Real-Time: Theory in Practice"*, number 600 in Lecture Notes in Computer Science, pages 1–27, 1992.

[2] Rajeev Alur, Costas Courcoubetis, and David Dill. Model Checking in Dense Real Time. *Information and Computation*, 104:2–34, 1993.

[3] Rajeev Alur, Costas Courcoubetis, Nicolas Halbwachs, Thomas A. Henzinger, Pei-Hsin Ho, Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. The Algorithmic Analysis of Hybrid Systems. *Theoretical Computer Science*, 138(1):3–34, 6 February 1995.

[4] Rajeev Alur and David Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 2(126):183–236, 1994.

[5] Tobias Amnell, Gerd Behrmann, Johan Bengtsson, Pedro R. D'Argenio, Alexandre David, Ansgar Fehnker, Thomas Hune, Bertrand Jeannet, Kim G. Larsen, M. Oliver Möller, Paul Pettersson, Carsten Weise, and Wang Yi. Uppaal - Now, Next, and Future. In F. Cassez, C. Jard, B. Rozoy, and M. Ryan, editors, *Modelling and Verification of Parallel Processes*, number 2067 in Lecture Notes in Computer Science Tutorial, pages 100–125. Springer–Verlag, 2001.

[6] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[7] Conrado Daws and Stavros Tripakis. Model checking of real-time reachability properties using abstractions. In Bernard Steffen, editor, *Proc. of the 4*th *Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, number 1384 in Lecture Notes in Computer Science, pages 313–329. Springer–Verlag, 1998.

[8] Conrado Daws and Sergio Yovine. Reducing the number of clock variables of timed automata. In *Proc. of the 17*th *IEEE Real-Time Systems Symposium*, pages 73–81. IEEE Computer Society Press, 1996.

[9] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HyTech: A Model Checker for Hybrid Systems. In Orna Grumberg, editor, *Proc. of the 9th Int. Conf. on Computer Aided Verification*, number 1254 in Lecture Notes in Computer Science, pages 460–463. Springer–Verlag, 1997.

[10] Torsten K. Iversen, Kåre J. Kristoffersen, Kim G. Larsen, Morten Laursen, Rune G. Madsen, Steffen K. Mortensen, Paul Pettersson, and Chris B. Thomasen. Model-Checking Real-Time Control Programs — Verifying LEGO Mindstorms Systems Using uppaal. In *Proc. of 12*th *Euromicro Conference on Real-Time Systems*, pages 147–155. IEEE Computer Society Press, June 2000.

[11] Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[12] Kim G. Larsen, Carsten Weise, Wang Yi, and Justin Pearson. Clock Difference Diagrams. *Nordic Journal of Computing*, 6(3):271–298, 1999.

[13] Sergio Yovine. Kronos: A Verification Tool for Real-Time Systems. *Springer International Journal of Software Tools for Technology Transfer*, 1(1/2):123–133, October 1997.