University of Twente

Formal Methods and Tools

Master's Thesis

Model Validation for Stochastic Hybrid Automata

Author:
Michiel Bakker

Supervisors:
Dr. Ir. Rom Langerak
Dr. Ir. André Kokkeler

August 24, 2020

Abstract

A hybrid automaton is a mathematical formalism used to model systems with both discrete and continuous behaviour. The theory of hybrid automata has been applied in a wide spectrum of research areas, including systems biology, automotive and avionic control systems and other cyber-physical systems. One of the available tools to model hybrid automata is Uppaal SMC, which uses a simulation-based method named statistical model checking to verify properties of a model. As models grow larger and more complex, it becomes more likely for an error to be accidentally introduced to the system. The process of confirming that a model is free of errors is called model validation. However, little research has been done on model validation tools for hybrid automata. In this research, we examined several methods for model validation, and put them into practice in a tool for Uppaal SMC. The validation methods were examined using theoretical analysis and evaluated using case studies, which indicated that the tool provides an effective way of performing model validation.

Contents

1 Introduction

Uppaal is a tool used for formal verification of timed systems. It has been used in the development of various applications, ranging from communication protocols to automotive systems [11, 13]. Uppaal is based on the theory of timed automata [2], and has support for features such as custom data structures and a specification language to aid in modelling. There are, however, limitations to the expressiveness of timed automata. Extending the formalism with behaviours such as variable clock rates make reachability checking of properties undecidable. This means that complex timed systems can only be approximated when modelled with Uppaal.

In order to support modelling of complex timed systems, an extension to Uppaal called Uppaal SMC was created. Uppaal SMC supports a more general model, where, among other extensions, the clock rates can be described using ordinary differential equations. Analysis of statistical properties of these models is done using *statistical model checking* (SMC). This is a simulation based approach, and can be seen as a compromise between testing and classical model checking. Uppaal SMC has been used in research areas such as systems biology, energy-centric systems, and recently, in modelling healthcare processes [6].

1.1 Problem Statement

Uppaal SMC can be effectively used to model complex timed systems. Analysis of the models using SMC can then be used to validate these systems. However, *system validation* is only effective when the model itself accurately models the system. Only a correct model can be used to verify properties of the system under test. Any tool used for system validation must therefore have ways for the modeller to find errors in their model, and to verify that the model is correct.

As models get larger and more complex, it becomes more likely for the modeller to make a mistake. In other programming and modelling applications this problem can be reduced using (automatic) analysis of the model, to spot errors early in the process. For example, research has been done on automatic validation of timed automata in Uppaal [14].

However, little similar research has been done on *stochastic hybrid automata*, the underlying formalism of Uppaal SMC. At this moment, there are no effective tools in Uppaal SMC that enable model validation. When the modeller suspects that there is an error in the model, one of the few ways to debug the program is to add extra debug variables and execute debug queries. This process can take a long time, and is also error-prone. There is therefore a clear need for additional methods to perform model validation of Uppaal SMC models.

1.2 Hybrid Automata

A hybrid automaton is a finite state machine extended with a finite set of real-valued variables. The values of the variables are described by ordinary differential equations. This combination of a discrete state (the nodes or 'locations' of the automaton) and a continuous state (the real-valued variables) allow the modelling of hybrid systems with both digital and analogue behaviours.

A simple example of a hybrid automaton would be a thermostat regulating the temperature of a room. This automaton could have two discrete states for the thermostat being either off or on. The temperature over time in either state can be modelled using differential equations derived from thermodynamics. Transitions between the discrete states can be restricted using expressions over the continuous state.

An effective method for the analysis of hybrid automata is stochastic model checking, where simulations (also called *runs*) are used to determine properties of the model. To achieve this, the definition of the model is extended such that non-deterministic transitions between states are replaced by probabilistic choices. This is the method used by Uppaal SMC to verify properties of its models.

A notable sub-class of hybrid automata are timed automata. A timed automaton can be seen as a hybrid automaton where all variables (usually called *clocks*) have a derivative over time of 1. In other words, all clocks are *uniform* and grow at the same rate. These restrictions make the reachability problem for timed automata decidable.

1.3 Model Validation

For this report, we define model validation in the context of formal verification as follows:

Model validation is the process of confirming that a model is correct.

It is interesting to note that this definition can apply to models in a very general sense. For example, consider a model in physics of material properties, like the friction of an object. Confidence that this model is correct can be gained by performing experiments, and confirming that the model accurately predicts the outcomes. Similarly, creators of software models must perform some form of validation to confirm correctness of the model.

In the field of formal methods, model validation is generally performed by algorithmically checking if the model conforms to a formal specification. This assumes that such a specification exists, and itself is complete and error-free. Intuitively, as the system to be modelled increases in complexity, so does its specification. This is especially true in timed systems, where behaviour of a model may depend on complex interactions between automata or precise timing constraints.

In this research, we consider the construction of a formal specification of a model, as well as its verification, part of model validation. This implies that model validation is equal parts a verification problem as a part of the development process.

1.4 Research Questions

This research attempts to find a solution to the problem stated above. In order to guide the research, the following research question is formulated:

In what ways is it possible to perform model validation of stochastic hybrid automata?

The research question is answered through the implementation of a tool for the model validation of Uppaal SMC models. It employs various methods to achieve this, which are evaluated using case studies. In order to further structure the research we define the following three sub-questions:

1. *Which properties of stochastic hybrid automata can be used to validate models?*
2. *How can properties of stochastic hybrid automata be verified?*
3. *How can the results of the validation best be communicated to the modeller?*

Question 1 is answered by defining several properties related to the validity of a model, and by implementing a method to verify these properties. The implementation and evaluation of two distinct verification methods is used to answer question 2. The evaluation of the validation properties and verification methods is done using case studies, where the tool is applied to models used in research. Here, errors in the models are attempted to be found and corrected using the tool, which is used to form an answer to question 3.

1.5 Contributions

This research examines and evaluates several methods to perform model validation for hybrid automata. These methods are implemented in a tool for Uppaal SMC, which is subsequently subjected to a qualitative evaluation. Due to the undecidability of the reachability checking problem for hybrid automata, several approximative methods for verification were examined. One of these methods is a novel technique where the hybrid automata is reduced to a timed automata, of which reachability checking is decidable. To our knowledge, this is the first time such a verification method has been applied to hybrid automata. Furthermore, it is also possible to perform verification using a simpler, simulation-based method.

1.6 Structure

The structure of this report is as follows. Section 2 provides an overview of the theory of timed automata and hybrid automata, and their respective implementations in Uppaal. Section 3 provides an overview of related research and its application to our research. The evaluation method and metrics are discussed in Section 4. Details of the implementation of the tool are discussed in Section 5, with the implemented verification methods and validation properties further elaborated on in Sections 6 and 7 respectively. The results of the evaluation are found in Section 8, and the report concludes in Section 9.

2 Background

2.1 Timed Automata

The formalism used by UPPAAL is based on the theory of timed automata [2]. A timed automaton (TA) can be seen as a finite state machine extended with a finite set of real-valued clocks. All clocks increase at the same rate, and transitions between states may or may not be taken based on predicates over clock values. This section will explain the basics of the theory of timed automata and its analysis.

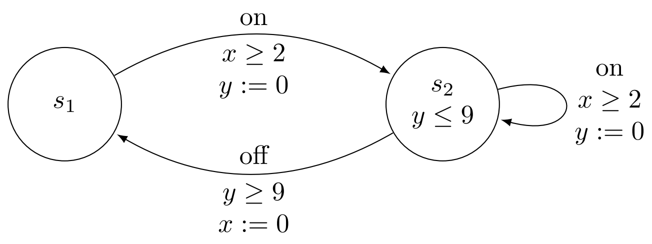


Figure 1: An example of a timed automaton modelling a light switch.

Example

Figure 1 shows an example of a simple timed automaton modelling a light switch. The light switch has two buttons, *off* and *on*. After 2 seconds of being off, the light can be switched on by pressing the *on* button. The light stays on until either the *off* button is pushed, or 9 seconds have elapsed. If the *on* button is pressed again before the 9 seconds have elapsed, the timer is reset and the light stays on for another 9 seconds. When the light turns off, it cannot be turned on again for two seconds.

The model has two locations, s_1 and s_2, two labels, *on* and *off* and two clocks, x and y. Location s_2 has the invariant $y \leq 9$. The two edges with the *on* label have a guard with the clock constraint $x \geq 2$ and $\{y\}$ as the set of clocks to be reset.

Syntax

A timed automaton is a tuple $A = \langle L, L^0, \Sigma, X, I, E \rangle$ where

- L is the finite set of locations.

  Note that in timed automata the term location is used instead of state. This is to avoid confusion, because the state of a timed automaton is described by the current location and by the valuation of the clocks.

- $L^0 \subseteq L$ is the set of initial locations.
- Σ is a finite set of labels.
- X is a finite set of clocks.
- $I : L \mapsto \Phi(X)$ maps every location to an invariant. The definition of a clock constraint $\Phi(X)$ is given below.
- $E \subseteq L \times \Sigma \times 2^X \times \Phi(X) \times L$ is a set of switches.

  A switch $(s, a, \phi, \lambda, s')$ is an edge from location s to s' with label a. ϕ is the guard that enables the switch, i.e. the clock valuation that is required to take the switch. λ is the set of clocks that is reset (valuation set to 0) when the switch is taken.

  Note again that the term switch is used instead of transition, as a transition in a timed automaton may change the current location as well as the valuation of the clocks.

Clock constraints are used as invariants of locations and guards of switches. For a set of clocks X, the set $\Phi(X)$ of clock constraints ϕ is defined by the grammar

$$\phi ::= x \prec c \mid x - y \prec c \mid \phi \wedge \phi$$

where clock $x, y \in X$, c is a constant in $\mathbb{N}$ and $\prec \in \{=, >, \geq, <, \leq\}$.

Transition System

A timed automaton can take two kinds of transitions: time transitions, where some amount of time passes, and switch transitions, where the discrete state changes.

Let S_A be the transition system of a timed automaton A. A state is noted as a pair (s, v), where s is a location in A, and v is a clock valuation. A clock valuation v is a mapping from the set of clocks X to the set of positive real numbers $\mathbb{R}^+$. For a delay $\delta \in \mathbb{R}^+$, let $v + \delta$ map every clock x to the value $v(x) + \delta$. A state of S_A is a pair (s, v), where v satisfies the invariant $I(s)$.

Delay transitions only affect the clock valuation, leaving the discrete state unchanged. They are a transition $(s, v) \xrightarrow{\delta} (s, v + \delta)$, where $\delta \geq 0$, and for all $\delta' \in [0, \delta]$, $v + \delta'$ satisfies $I(s)$.

With a switch transition the discrete state changes, while no time passes. Given a switch $(s, a, \phi, \lambda, s')$ and a state (s, v) where v satisfies ϕ, the transition is noted $(s, v) \xrightarrow{a} (s', v[\lambda = 0])$. $v[\lambda = 0]$ denotes the clock valuation where every clock $x \in \lambda$ is assigned the value 0, and the other clocks are unchanged.

As an example, the following transition sequence is possible in the automata of Figure 1. Here, four delay transitions and three switch transitions are taken.

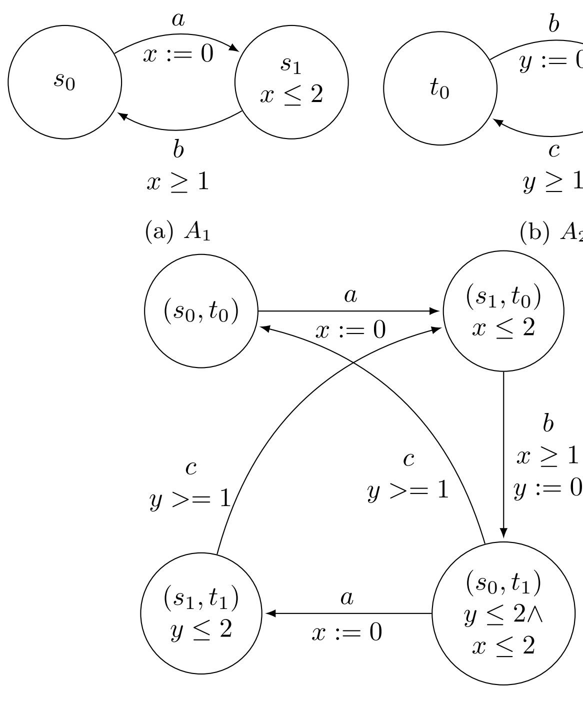
$(s_1, [x = 0, y = 0]) \xrightarrow{1}$

$(s_1, [x = 1, y = 1]) \xrightarrow{2}\xrightarrow{\text{on}}$

$(s_2, [x = 3, y = 0]) \xrightarrow{1} \xrightarrow{\text{on}}$
$(s_2, [x = 4, y = 0]) \xrightarrow{9} \xrightarrow{\text{off}}$
$(s_1, [x = 0, y = 9])$

Product construction

A particularly useful feature of the analysis of timed automata is the product construction of two automata, also known as parallel automata. This allows a large and complex model to be made by composing multiple smaller models. Given two timed automata A_1 and A_2, the product construction is noted by $A = A_1 \parallel A_2$. The set of (initial) locations of A is the cartesian product of the (initial) locations of A_1 and A_2. Switches that share a common label are synchronised. The guard of the synchronised switch is the conjunction of the two guards, and the set of clocks to be reset is the union of the clock resets of the two switches.



(a) A_1 (b) A_2

(c) $A_1 \parallel A_2$

Figure 2: Two timed automata and their product construction.

Figure 2 shows an example of the product construction of two automata. Note that the set of locations of $A_1 \parallel A_2$ is the cartesian product of the locations of the two automata. A_1 and A_2 both have a switch with label b, so these two switches are synchronised.

Model checking

In model checking applications, the specification of the model is often formalised using CTL. A CTL formula can, for example, specify that some property p must hold in all reachable states, in at least one state or hold until some other property q holds. *Timed Computation Tree Logic* (TCTL) is an extension of CTL where the temporal operators can have time bounds [1]. With TCTL it is possible to express properties such as "property p always leads to property q within 3 time units".

A brief overview of the subset of TCTL used in this report is given here. TCTL has two quantifiers over paths, $A\phi$ and $E\phi$, where ϕ must hold on *all* paths or *any* path from the current state respectively. Path formulas are of the form $\Box\phi$ or $\Diamond\phi$. With $\Box\phi$, ϕ holds on all states of the current path and $\Diamond\phi$ notes that ϕ holds in any state of the path.

Clocks in a TA are real-valued, causing the transition system of a model to have an infinite set of states and transitions. Because of this, model checkers for TA must represent the model abstractly in order to have a decidable algorithm to verify logical properties. This is done by transforming the timed automaton to a *region automata*, which can represent the infinite set of clock values in a finite set of states [2].

2.2 Timed Automata in UPPAAL

A model in Uppaal is a Network of multiple Timed Automata (NTA) in parallel. To facilitate modelling the model is extended beyond the pure timed automata formalism with a couple of extra features [3]. One of those features is the inclusion of bounded discrete variables that are part of the state. These variables can be read, written to and can be used in guards and invariants.

Uppaal terminology varies slightly from the standard terminology of timed automata. For instance, switches are usually called edges, an automaton is often called a process, and the term channel is used instead of label. An edge can be *fired*, meaning that the system is in a state where that switch transition can be taken. This report will in most cases use the terminology used in Uppaal literature instead of standard TA terminology.

Discrete Variables

Uppaal extends the TA formalism by adding bounded discrete variables that are part of the state. These variables can be used in expressions in guards and invariants, and their values can change according to an update expression. Uppaal also includes an extensive specification language that is used to observe and change the discrete state of the model. This language is based on C, and allows the user to model complex behaviours.

The discrete state is shared across all automata in the model. This allows it to be used to transfer data between processes. The sending process can (temporarily) assign values to discrete variables, which can be read by the receiving process. This can, for instance, be used to model communicating nodes in a computer network.

UPPAAL models

In UPPAAL channels are declared using the syntax `chan c`. Two processes can synchronise if one has an edge label with `c!` enabled, and other an edge with `c?`. If the channel is marked as a broadcast channel (using the syntax `broadcast chan c`), the sending edge labelled `c!` can synchronise with zero or more receiving edges labelled `c?`. An edge with label `c!` can still fire even if no edge with `c?` is enabled, which means that sending a broadcast is not blocking.

A location in a UPPAAL system can be marked as urgent. When one of the automata in the system is in an urgent location, no time is allowed to pass. A location can also be marked as committed, which is similar but even more restrictive. When one or more of the locations in the state is committed, no time is allowed to pass, and the next transition must involve an edge moving out of a committed location.

Syntax and Semantics

We use the following syntax for UPPAAL models. Let $A_i = \langle L_i, l_i^0, \Sigma, X, I_i, E_i \rangle$ be an network of n timed automata where

- L_i is the set of locations for TA A_i.
- l_0^i is the initial location vector.
- $\Sigma = \Sigma_! \cup \Sigma_?$ is the set of labels (or channels), where $\Sigma_!$ is the set of *sending* labels, and $\Sigma_?$ the set of *receiving* labels.
- X is the set of clock and data variables.
- I_i is the mapping from a location to an invariant expression.
- E_i is the set of edges.

  A switch (l, a, g, u, l') is a switch from location l to l', with label $a \in \{\tau\} \cup \Sigma$, guard expression g and update expression u. An update expression is a set of expressions in the form $v := e$ with $v \in X$ and e an expression if v is a data variable, or 0 if v is a clock.

For an expression e and valuation v, we write $v \models e$ to mean that the expression is satisfied with the given valuation. $v[u]$ denotes the valuation v changed according to update expression u. Given a location vector $\bar{l}$, we use $I(\bar{l})$ to mean the conjunction of all invariants $I_i(l_i)$ in $\bar{l}$. We write $\bar{l}[l_i'/l_i]$ to mean the location vector where location l_i is replaced with location l_i'.

A state in a NTA is a pair $(\bar{l}, v)$ with a location vector l, and valuation v mapping clocks and data variables to their current values.

The definition of a delay transition remains mostly unchanged, being a transition $(\bar{l}, v) \xrightarrow{\delta} (\bar{l}, v + \delta)$ with $\delta \in \mathbb{R}^+$ and for all $\delta' \in [0, \delta]$, $v + \delta' \models I(\bar{l})$

The definition of switch transitions are changed to account for blocking synchronisations and changes to the discrete variable valuation. We distinguish two cases:

- Internal transitions $(\bar{l}, v) \xrightarrow{\tau} (\bar{l}[l'_i/l_i], v')$ iff there exists an edge (l_i, τ, g, u, l'_i) where $v \models g$ and $v' = v[u]$.
- Binary synchronisation transitions $(\bar{l}, v) \xrightarrow{\tau} (\bar{l}[l'_i/l_i, l'_j/l_j], v')$ iff there exists edges $(l_i, c!, g_i, u_i, l'_i)$ and $(l_j, c?, g_j, u_j, l'_j)$ where $i \neq j$, $v \models (g_i \wedge g_j)$ and $v' = v[u_i \wedge u_j]$.

Templates

Processes are instantiated from templates as defined in the system declaration. A template can have zero or more parameters of any type. These parameters can be referenced anywhere in the model, and are replaced by their actual argument as defined in the system declaration. Templates are a useful feature for models with a number of almost-equal automata, such as a train-gate controller with n trains.

Queries

The query language used by UPPAAL is a simplified version of TCTL where nested path formulas are not allowed. Queries fall in three different classes: reachability, safety and liveness properties. Reachability properties can be seen as questions in the form "does property p hold *eventually*?" and are written using the syntax `E<>` p. A safety property states that some property *always* holds, and are usually formulated to say that something bad never happens. The query `A[]` p is satisfied when p holds in all reachable states. Liveness properties state that something good will eventually happen. A useful operator for specifying liveness properties is the *leads to* operator p `-->` q, which states that whenever p is satisfied, q is eventually satisfied.

State formulas are boolean expressions over the discrete and clock variables of the models, location expressions and the special `deadlock` expression. Location expressions are of the form `Process.Location`, and evaluate to true if the specified location is in the location vector of the current state. The `deadlock` expression is only allowed in symbolic queries. It evaluates to true for state $(\bar{l}, v)$ if for all $\delta \geq 0$, $(\bar{l}, v + \delta)$ has no enabled action transition.

2.3 Stochastic Timed Automata

Timed automata can be effectively used for the modelling and analysis of cyber-physical systems. But even though the formalism is powerful, many real world complex systems depend on complex dynamics or stochastic behaviour. These systems can only be approximated when modelled using TA. Furthermore, the computation time of model checking a TA is heavily dependant on the size of the state space. Extensions to the model can lead to an exponential increase of computation time. This can make the analysis of models with a large state space practically infeasible.

In answer to the above problems a new formalism, *stochastic timed automata*, is proposed. STAs are an extension to the timed automata formalism, where stochastic semantics are added to the model. The transition system of an STA remains unchanged from its TA counterpart. The non-deterministic choices of delay and switch transitions of TA are instead interpreted as probabilistic choices. The added semantics allows for a richer analysis of the model. Instead of asking whether a state is reachable, it is now possible to ask what the *probability* of reaching a state is. Networks of STAs (NSTA) are created by composing multiple STAs. Stochastic semantics for NSTAs are based on races between components, and are used to efficiently generate simulations of the model.

The stochastic interpretation of the model allows for analysis using *statistical model checking* instead of traditional model checking methods. SMC uses simulations of the model to test a hypothesis, and uses the outcome of these simulations as the statistical data for quantitative properties of the model. Because SMC is based on runs instead of searching the state space, it can be vastly more efficient in terms of time and memory consumption compared to exhaustive model checking methods. The rest of this section describes the stochastic semantics of STAs and details of SMC methods for NSTAs. Full formal definitions and semantics for NPTAs (of which NSTAs are a subset) can be found in work by David et al. [5].

Delay distributions

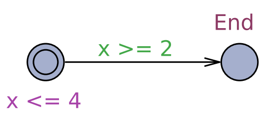
With the stochastic interpretation of a TA, delays are chosen from a probability distribution. Two cases to be distinguished: bounded or unbounded delay. If the invariant of the current location of the automaton contains an upper bound on a clock, that location is *bounded*, and a delay is chosen from a uniform distribution up to the limit of said constraint.

If the location has no upper bound, the delay is chosen from an exponential distribution. The model is extended with a function $P : L \mapsto \mathbb{R}^+$ mapping a location to an exponential rate. The probability density function of delay $\delta \in [0, \infty)$ is $F(\delta) = \lambda e^{-\lambda\delta}$. During run generation, the concrete delay is generated by $\frac{-ln(u)}{\lambda}$, where u is a uniform random real from the interval $(0, 1]$.

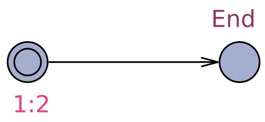
For example, the delay of the transition to location End in Figure 3a is chosen with an uniform distribution from the interval $[2, 4]$. Similarly, the delay in Figure 3b is chosen from an exponential distribution with rate $\frac{1}{2}$.

Statistical Model Checking

The basis of SMC approaches for NSTAs is an algorithm that generates runs of a model, an outline of which is given here. Inputs to the algorithm are an NSTA A, consisting of automata $A_1...A_n$, a clock bound $c \in \mathbb{R}$ and a clock C



(a)



(b)

Figure 3: STAs with a bounded and an unbounded delay.

of automaton A. Runs are generated until the value of the clock $v(C) \geq c$. Starting from the initial state $(\bar{l}_0, v_0)$, let the current state be noted as $(\bar{l}, v)$. Each automaton picks a delay $\delta_i \in \mathbb{R}^+$ as described in the previous section. The automaton A_k with the lowest delay δ_k is the 'winner' of the race. If multiple automata have delays $\delta_i = min_{1 \leq i \leq n}(\delta_i)$, an automaton is chosen uniformly out of the possible automata. A switch a is chosen uniformly from the set of enabled switch transitions in state $(\bar{l}, v + \delta_k)$ [1]. The transitions $\xrightarrow{\delta_k} (\bar{l}, v + \delta_k) \xrightarrow{a} (\bar{l}', v')$ are added to the run, and $(\bar{l}', v')$ is assigned to the current state. This process is repeated until $v(C) \geq c$.

Runs of the model are used as the statistical evidence for SMC queries. Probabilistic properties are expressed using the following syntax:

$$\psi ::= \mathbb{P}(\Diamond_{C \leq c} \phi) \mid \mathbb{P}(\Box_{C \leq c} \phi)$$

Where C and c are clock bounds as described above and ϕ a state property. The expression denotes the probability that property ϕ is satisfied in either at least one state of a run, or in all states of a run respectively. The notation is a fragment of Probabilistic CTL (PCTL). Full semantics of PCTL applied to NSTAs can be found in work by David et al. [5]. Since $\mathbb{P}(\Box_{C \leq c} \phi) \equiv 1 - \mathbb{P}(\Diamond_{C \leq c} \neg\phi)$, SMC algorithms only need a procedure to compute if the property $\Diamond_{C \leq c} \phi$ holds for an NSTA.

SMC queries can be either qualitative or quantitative. In the qualitative case, the probability $p = \mathbb{P}(\Diamond_{C \leq c} \phi)$ is compared against some bound $b \in [0, 1]$. In the quantitative case, the value of p is approximated. In both cases, the user supplies a number of parameters describing the interval in which p may lie, and the allowable chances of false or true positives. These parameters are used to determine the number of runs needed for the computation.

2.4 Stochastic Hybrid Automata

The formalism of timed automata is quite restricted because in general even small extensions to the formalism (such as variable clock rates) lead to the loss of decidability of reachability algorithms. However, because SMC uses simulations of the model instead of reachability algorithms, the restrictions of TA can be relaxed. If the formalism is extended such that clock rates and clock resets can

[1]This is possible because a valid NSTA is assumed to be *input-enabled*, i.e. in every state a sending switch transition is possible.

be specified by an arbitrary expression over clocks, the formalism is generalised to that of *hybrid automata* [8]. In this report, we refer to hybrid automata with defined stochastic semantics as *stochastic hybrid automata* (SHA).

Standard terminology for hybrid automata differs slightly from that of timed automata. For instance, the equivalent of clocks in HA are called variables, locations are control modes and guards are jump conditions. For clarity, this reports uses timed automata terminology over HA terminology wherever applicable.

Hybrid automata is a powerful formalism suitable for the modelling of many systems with both discrete and complex continuous dynamics. The reachability problem of HA is not decidable in the general case, except for more limited sub-classes such as initialized rectangular hybrid automata.

Syntax and Transition System

As with timed automata, the state space of a hybrid automata A is every pair (s, v) where the clock valuation v satisfies the invariant $I(s)$ of location s. The notation of clock delay $v + \delta$ with $\delta \in \mathbb{R}^+$ is extended to mean all clocks updated according to clock rates in the current location.

With the generalised clock delay function, the definition of delay transitions remains unchanged. Instead of only resetting clocks, switch transitions can assign any real value to the clocks according to an expression over clocks.

Analysis

Model checking tools for hybrid automata are limited, but available. An early tool for the analysis of HA is HyTech [9]. It is a symbolic model checker, and limited to the class of *linear hybrid automata*.

To support the analysis of general hybrid automata, SMC techniques must be used. Uppaal SMC is a notable tool capable of this. SMC algorithms for (networks of) HA are similar to those of NSTAs. The difference is in the run generation algorithm, which must be updated to support the richer formalism of HA. Since clock rates are defined by general expressions over clocks, the rate may be described by an ordinary differential equation (ODE). Thus, the clock valuations after a delay transition must be computed using an ODE solver. Uppaal SMC uses fixed time step Euler's integration method. Note that this only computes an approximation of the value of the clock, and results are sensitive to the time step configured by the user.

2.5 Hybrid Automata in UPPAAL SMC

This section provides an overview of the extended formalism of Uppaal SMC and its analysis methods. When verifying SMC queries, the Uppaal formalism is extended to support the modelling of stochastic hybrid systems by allowing clock rates to be defined by general expressions over clocks. Additionally, Uppaal SMC models support branches with weighted edges to model probabilistic behaviour on discrete state changes.

Branches

When multiple switch transitions are enabled in a TA model, the choice of which transition to take next is non-deterministic. In an NSTA, this non-deterministic choice is replaced by a probabilistic choice over the currently enabled edges. Uppaal SMC allows the modeller to define *branches*, where outgoing edges are labelled with a weight. If a switch transition has multiple enabled branches, the chance that branch i is taken equals $w_i/\Sigma_j w_j$ with w_i being the weight of branch i and $\Sigma_j w_j$ the sum of all enabled branches. Weights of branches are defined by general expressions over the variables, and may thus change based on the state of the model. As an example, the branch to `L1` in Figure 4b is taken with a probability of $\frac{1}{4}$.

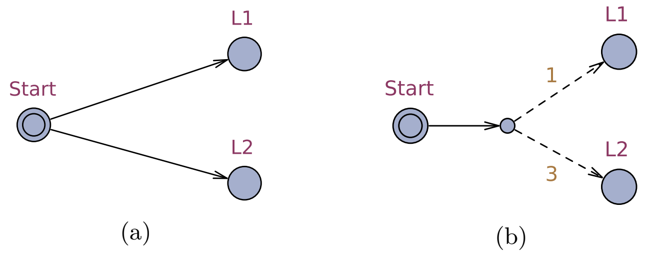


Figure 4: A TA with a non-deterministic choice and its STA counterpart.

Clock rates

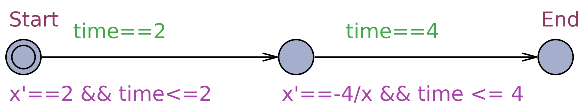
In an NHA model, the rate at which clocks evolve may vary from each other. These clock rates can be defined by simple constants, or by general expressions involving clocks, allowing clock valuations to be defined by ordinary differential equations.

Figure 5a shows an example of an HA with variable clock rates, with Figure 5b showing the result of a simulation of the HA. While in the `Start` location, clock x increases by 2 units per time unit elapsed. After two time units, the middle location is reached, and the value of x is defined by the differential equation $x' = \frac{-4}{x}$.

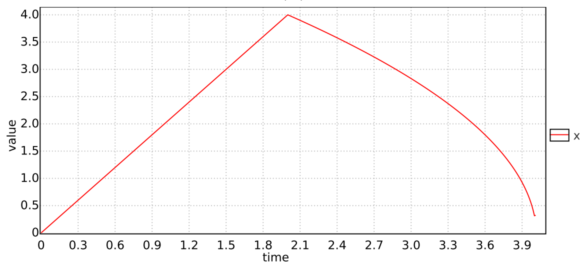
Query language

Uppaal SMC allows for the checking of queries related to the stochastic interpretation of timed automata. Instead of verifying whether some property is satisfiable, the additional queries allow for reasoning about the *probability* that some property holds. These results of these queries are approximated using statistical model checking algorithms. This means that no exact answer is given. Instead, an answer with a certain confidence is given.

For example, if we refer back to Figure 3a, we can ask what the probability p is that after 3 time units the system is in the `End` location. The probability



(a)



(b)

Figure 5: An HA with variable clock rates and the result of its simulation.

estimation algorithm computes how many runs are needed to produce an interval $[p - \varepsilon, p + \varepsilon]$ with confidence $1 - a$, where ε and a are user-provided constants. Here, the term confidence is used to mean the chance that the true probability of the query lies in the interval $[p - \varepsilon, p + \varepsilon]$. In UPPAAL SMC syntax, this is expressed using the following query:

```
Pr [<=3] (<> Process.End)
```

A result of this query is that p is in $[0.46857, 0.568559]$ with a confidence of 0.95. The statistical model checking algorithm determines the amount of simulations required to compute an answer with the specified confidence a and precision ε on the fly. In this case, 401 simulations were needed.

Additionally, it is possible to perform hypothesis testing, where the user asks the question if the probability of a certain property is greater or equal than a certain value. Also it is possible to test whether the probability of one property is greater than the probability of a second property.

Synchronisation

Synchronisations in UPPAAL SMC are restricted to broadcast synchronisations. This means that synchronisations are never blocking, and that one process can synchronise with zero or more processes. Multiple processes can synchronise using channels and communicate using shared variables to generate a Network of Stochastic Hybrid Automata (NSHA).

Floating-Point support

An additional type is added to the syntax to allow for double precision floating-point variables. Variables of this type can be used in general expressions and used to store arithmetic expressions. When the clocks are declared as `hybrid clock` and the floating-point variables are not used for control of the automata (i.e. not used in guards, invariants etc.) model checking of the system is still possible. This means variables that track costs are able to be modelled while still being able to model check the system.

3 Related Work

3.1 Validation of Timed Automata

Work by Onis [14] presents an approach for model validation of timed automata. The research was limited to the validation of 'pure' Uppaal models, so Uppaal SMC models were not considered. As part of the research, a tool called UrPal was developed that automatically performed sanity checks of Uppaal systems in an effort to spot errors early in the modelling process. The tool supports a number of checks, including unwanted deadlock detection and reachability analysis for all locations and edges in a system. The GUI of Uppaal was extended to easily allow the modeller to execute the checks and intuitively show any potential errors in the editor.

The quality of the sanity checks were evaluated using four metrics: soundness, completeness, efficiency and effectiveness. Soundness meant that all reported errors were actual errors, i.e. no false positives are shown to the user. Similarly, completeness means the absence of false negatives, meaning that if the model contains any errors, they are all reported. Efficiency was used to mean the combined efficiency in both time and memory space. Lastly, effectiveness indicates that the result of the check contains enough information to find the error.

The research highlights some important qualities of validation tools. Especially the presence of false positives is very detrimental to the effectiveness of a checker. This is because when a user encounters multiple false positives from the tool, they no longer trust any of the results and starts ignoring them. Also efficiency of the check is important, because the longer a check takes, the less likely the modeller is to perform it regularly.

The tool was implemented as a plug-in and allows the modeller to intuitively find the error by extending the Uppaal editor. For example, unreachable locations and edges are shown to the user by colouring them differently in the editor. Also, if an unwanted state (such as a deadlocked state) is reachable, it is possible to load a trace to this state into the editor, allowing the modeller to find the cause of the error.

The plug-in that was developed was used as the basis of the implementation for the model validation methods explored in this research. More specifically, techniques for model transformations used to verify properties in UrPal were used for properties of Uppaal SMC models in our research. Furthermore, a method was explored to adapt the verification of the sanity checks to be applicable to Uppaal SMC models.

3.2 Abstraction Refinement

Counterexample guided abstraction refinement (CEGAR) [4] is a technique for the model checking of systems with large state spaces. Model checking these systems with traditional methods is infeasible because of state space explosion. The solution that CEGAR offers is to first create an abstraction of the original model that over-approximates the original behaviour. This abstracted model has a smaller state space, which enables regular model checkers to verify properties of it. If the abstracted model satisfies a property, then the concrete model also satisfies the property. If not, the counterexample produced by the model checker is used for further analysis. If the counterexample is also a counterexample for the concrete model, the property is unproven. If not, the counterexample is *spurious* and the process continues. As long as the property is not proven or unproven the abstracted model is continuously refined using information from the counterexample.

In research by Dierks et al. the CEGAR technique is applied to timed automata model checking using UPPAAL [7]. In their work, the model is abstracted using *variable hiding*. In the coarsest abstraction, all variables are hidden, and with every abstraction refinement loop one or more variables are removed from the list of hidden variables. Counterexamples are analysed by transforming the trace to a linear test automaton, and model checking the composition of the test automaton and the concrete model.

A similar approach is used in research by Jha et al. [12]. Their research proposes a method for reachability analysis of linear hybrid automata inspired by the CEGAR technique. Their abstraction technique is similar to that of Dierks et al., in that first all continuous variables are hidden, and with every refinement loop a set of variables is chosen to be added to the set of used variables.

We cannot directly apply the CEGAR technique for the validation of HAs because the reachability problem for the model is in general undecidable. Instead, the abstraction technique using variable hiding is used as the basis for the HA to TA abstraction method explored in this research. In our method, the abstraction is not used to reduce the state space, but is instead used to transform a model from an undecidable formalism (HA) to a decidable one (TA) by hiding clock variables with a non-uniform rate. While not explored in this research, the method for verifying whether a counter-example is spurious through the use of a test automaton could be applied to check if a trace obtained from the abstract TA model is part of the language of the original model.

3.3 UPPAAL Slicing

The implementation of variable-hiding abstractions for timed automata, as described in the previous section, is relatively straight-forward using reductions of expressions in guards and invariants. Doing the same for UPPAAL models introduces complications, since UPPAAL extends the TA formalism by (among other features) a C-like modelling language. The valuation of a hidden variable can affect other variables in the model depending on the control flow of a function. Automatic analysis of the modelling language to determine the dependencies of such variables is a non-trivial problem.

Work by Sørensen and Thrane [16] introduces a method for automatic *program slicing* [18] of Uppaal models. Program slicing is the (automated) method of reducing a program to use a subset of its variables while preserving (a part of) the behaviour of the original program. The work describes a program slicing technique for Uppaal models that preserves reachability.

The intended use of the method was to reduce the state space for models, in order to make verification practically possible, where previously it was not. While the method was not used in our research, it could be applied in conjunction with the variable hiding abstraction method to produce a model with a minimal abstraction of the discrete variables.

4 Method

Recall the research question: *In what ways is it possible to perform model validation of stochastic hybrid automata?*. This research attempts to answer this question by bringing potential model validation techniques into practice, by implementing them as part of a tool for Uppaal SMC models. Next, the techniques are evaluated in two ways. Firstly, a theoretical analysis of the verification methods is made to determine whether the procedures are sound and complete (i.e., unable to produce false positives or false negatives). Secondly, a qualitative evaluation of the tool is performed using case studies. The cases use Uppaal SMC models used in research, and attempt to replicate the model validation process with the aid of our tool in a real-world context.

4.1 Tool Design

The tool is designed so that the three sub-research questions can be answered. The first question *'Which properties of stochastic hybrid automata can be used to validate models?'* is answered in Section 7. It describes all implemented model validation properties in detail. The practical utility of these properties as a tool to specify a valid model is explored in the case studies.

Section 6 answers the second question, *'How can properties of stochastic hybrid automata be verified?'* by discussing two implemented algorithms. The architecture separates the verification algorithms from the model specification so that it becomes possible to compare them. Soundness and completeness of the verification methods for properties of hybrid automata are determined. A qualitative evaluation is performed as part of the case studies, the results of which are used to make a comparison of the merits and demerits of both methods.

The final research question, *'How can the results of the validation best be communicated to the user?'*, is answered through implementation in the tool and evaluation through case studies. The design of the user interface is such that it allows the user to perform two aspects of model validation: finding faults and verifying the absence of faults. The evaluation assesses the effectiveness of this process by determining the use of the tool in finding and fixing faults in a model.

4.2 Evaluation Method

In the case studies, the tool is used for the validation of Uppaal SMC models used in research. For each model, a specification is made which is consequently verified. The result of this is either that the model is valid (according to the specification), or that it contains faults. If the model contains a fault, the tool is used to alleviate the problem, and the steps required to produce a valid model are noted. To structure the evaluation, several qualitative metrics for the properties used for validation and the verification methods. For the verification methods, we use the following metrics:

- **Soundness** – If the outcome of the procedure is that a model is valid, then the model is valid in truth. In other words, the procedure does not produce false positives.
- **Completeness** – If a model is valid, then the outcome of the procedure is that the model is valid. In other words, the procedure does not produce false negatives.
- **Time performance** – The time required for verification is measured. Since these measurements can be highly dependent on characteristics of the model (e.g. number of locations, edges, variables et cetera) the individual measurements are not very meaningful. Instead, the measurements are used to determine the practicality of the method. For instance, a procedure might produce correct outcomes, yet if the computations takes over a day its practical utility is limited.
- **Utility** – The above metrics are used to make an assessment of the utility of the verification method in the following three use cases: proving validity, finding faults and fixing faults. This can be used to show the specific strengths of a method in a particular use case.

For the evaluation of the validation properties we use the following qualitative metrics:

- **Ease of use** – A measure of how easy it is to configure the property to accurately specify correct behaviour of the model.
- **Improvement over existing methods** – What alternatives to this property exist, and how much of an improvement is the property over said alternatives?
- **Applicability** – What is the range of use cases where the property is helpful?

5 Implementation

In order to evaluate model validation methods a plug-in for the Uppaal editor was developed. When installed, an additional tab is present in the editor, in which the modeller can configure the model validation procedure. This configuration can be manipulated using an user interface that visualises all available options. Alternatively, this configuration can be specified with a textual format, and verification can be performed using the command line. This option can be used to automate the model validation procedure. The rest of this section will go into further detail of the internal implementation of the tool.

5.1 Architecture

5.1.1 Model views

As mentioned above, the tool is mainly intended to be used as a plug-in for Uppaal. This choice was made because it made for a user-friendly way of installing and using the tool (requiring only the user to place a file in the Uppaal installation directory). Furthermore, all of the various options for validation are laid out in the GUI, allowing the modeller to try out various methods without having to resort to reading a manual every time.

The tool is a heavily adapted version of the UrPal tool developed by Onis [14]. Re-using this existing code base allowed for faster development, since it already provided practical ways of manipulating the Uppaal editor, parsing and transforming models and verifying queries on (transformed) models.

$$\begin{tikzcd}
\boxed{\text{Plain text}} \arrow[r, leftrightarrow] \arrow[d, leftrightarrow] & \boxed{\texttt{NSTA}} \\
\boxed{\texttt{Document}} \arrow[r] & \boxed{\texttt{UppaalSystem}}
\end{tikzcd}$$

Figure 6: Representations of Uppaal SMC models and their transformations.

Figure 6 gives an overview of the various representations of Uppaal SMC models used internally. Plug-ins can read and modify models loaded in the editor as a `Document` object. This view is a thin wrapper over the XML format of models, and also includes 'stylistic' information such as positions of locations, comments and node colours. Since this is a direct view of the model loaded in the editor, invalid models (e.g. models with syntax errors) can also be represented using this format.

A `Document` object can be compiled to an `UppaalSystem` object. Here, the templates are instantiated into zero or more processes and the specifications

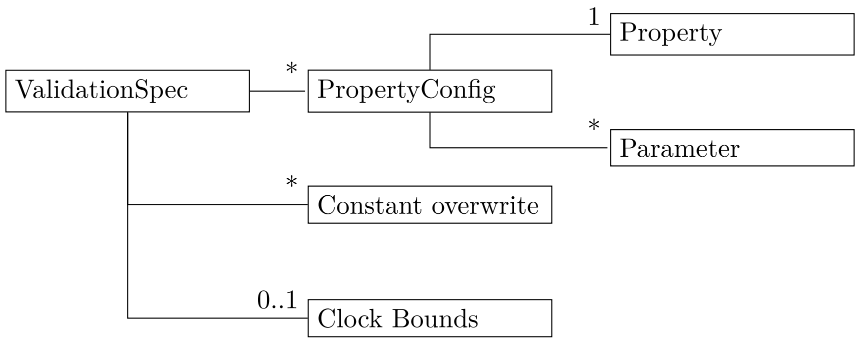


Figure 7: Simplified class diagram of the validation specification.

and expressions are compiled to an intermediate format. Naturally, all stylistic information such as text formatting and node positions is lost in the compilation process. A successfully compiled `UppaalSystem` object can be used for verification of applicable UPPAAL queries.

In a `Document` object, all expressions and declarations of a model are stored as plain text. In order to effectively manipulate the loaded models they are first parsed to an `NSTA` [2] object. This interface is provided as a library with Java code generated from EMF [17] metamodels, developed by Schivo et al. [15]. The library represents UPPAAL models in an object-oriented way, with expressions and the specification language being parsed to an AST. Using the library facilitates writing model transformations, since writing a custom parser for UPPAAL syntax is avoided, and any model generated from an `NSTA` object will already be syntactically correct.

The transformations between UPPAAL XML files and `NSTA` objects (and vice-versa) only preserves structural information. Any comments or special formatting in the expressions and specification language are lost because of the conversion to an AST. Furthermore, layout information such as positions of locations, edges or labels and location colours is lost in the parsing and serialisation process.

5.1.2 Validation Specification

The plug-in allows the modeller to create a simple specification for their model, called the *validation specification.* The modeller constructs this specification such that when all checks pass, the model is considered valid. This is a different approach to the interface of validation tools when compared to UrPal. Whereas UrPal provided a list of checks that required no configuration at all, our im-

[2]The class as used by the library is called `NSTA`. This is a slight misnomer since the model supports clock rates defined by expressions over clocks, and thus is a hybrid automaton instead of a (stochastic) timed automaton. This report refers to the *class* `NSTA` using a monospaced font, being distinct from the formalism of Networks of Stochastic Timed Automata (NSTA).

plemented validation methods requires configuration of the checks according to the desired behaviour of the model. This is a compromise in the user-interface philosophy, trading ease and speed of setting up the tool for more powerful verification methods.

The specification consists of a sequence of checks combined with a configuration for model checking and a pre-processing step. Figure 7 shows a partial class diagram of the validation specification. A `ValidationSpec` contains a list of `PropertyConfig` objects, which have a reference to the actual property being checked (further explained in Section 7) and its parameters. The parameter field is a key-value mapping containing both mandatory and optional parameters, depending on the property. A `ValidationSpec` optionally has clock bounds, and zero or more constant variable overwrites, both related to pre-processing steps. These options are further elaborated on in section 6.2.3 and 6.2.4 respectively.

The validation specification can be parsed from and serialised to a JSON format. When using the plug-in, this allows the specification to be saved with the UPPAAL model, preserving the specification and allowing it to be easily shared.

5.2 Separation of Model Checker and Property

As the reachability problem for hybrid automata is not decidable, some approximative method must be developed to verify properties of the model. This research investigated two methods, a symbolic and a concrete checking method, which are further elaborated on in section 6.

Furthermore, the tool uses several *validation properties*. Conceptually, a validation property is a yes/no question over the model, with a positive answer indicating validity. Section 7 goes into further detail on the implemented properties.

For maximal flexibility, the tool can pair both model checking methods with any validation property. Figure 8 gives an overview of the verification method for the validation properties. First, a number of pre-processing steps are applied to the model, related to reducing the verification time and restricting model behaviour (further discussed in sections 6.2.3 and 6.2.4). A validation property is abstracted to a function that takes an UPPAAL SMC model as parameter, and returns a transformed model, and a state formula. The abstract view of a reachability checker is a function with an UPPAAL SMC model and a state formula as parameters, which returns whether the model can reach a state satisfying the given state formula. Optionally, a trace leading to such a state is produced, as illustrated in Figure 9. This abstraction allows the verification algorithm to be agnostic of the reachability checking method used, and thus easily allows the modeller to choose the method that suits their use case optimally.

5.3 UPPAAL Editor Integration

Manipulation of the UPPAAL editor is done using the recently supported plug-in API. This API allows the currently loaded model to be read and modified

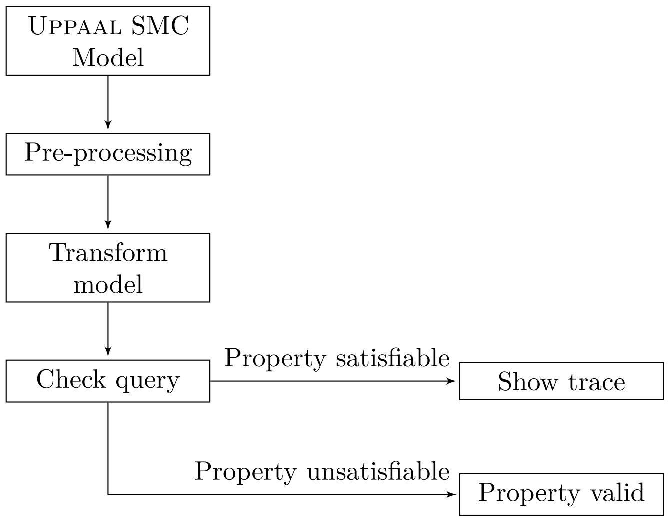


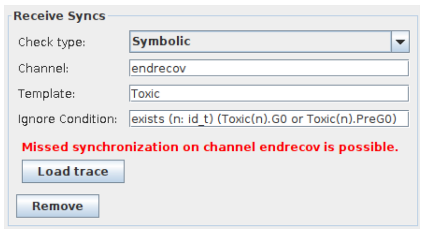
Figure 8: Schematic view of the verification method for validation properties.

using the `Document` interface. Additionally, traces obtained from the verification of UPPAAL queries can be loaded in the editor. However, only the loading of symbolic traces is currently supported. These traces are tightly coupled to the currently loaded model in the editor, which causes traces obtained from transformed models not to be easily loaded in the editor without additional transformations. In order to be shown in the editor, traces from transformed models must be modified such that the general structure (number of variables, template instantiations et cetera) matches the original loaded model. This is the approach that was used for one of the sanity checks implemented in UrPal.

Since the intended usage of our tool involves showing both symbolic and concrete traces, and these traces are obtained from queries over transformed models, loading traces using the plug-in API is circumvented. Instead, traces are shown using manipulation of UI elements in the editor. Effectively, this emulates manually loading the transformed model, adding a query to the query list, and pressing the button to show a trace in the editor.

This alternative approach has the benefit of working with any transformed model, without the need to transform the trace. Also, concrete traces are supported without requiring any additional work. The drawback to this approach is that trace view in the editor shows the transformed model, instead of the original model. This causes auxiliary variables required for property verification to be visible to the user. The formatting information of expressions (guards, invariants et cetera) is lost in the conversion of a model to a `NSTA` view and back. This causes the model shown in the trace view to differ visually from the original model, and may lead to overlap of expression labels.

The `NSTA` model view does not contain the positions of locations, location


Receive Syncs

Check type: **Symbolic**

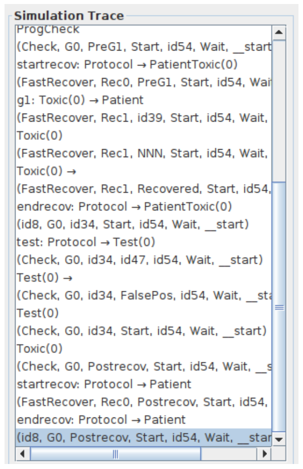
Channel: endrecov

Template: Toxic

Ignore Condition: exists (n: id_t) (Toxic(n).G0 or Toxic(n).PreG0)

Missed synchronization on channel endrecov is possible.

(a) UI component of an unsatisfied validation property. Pressing the 'Load Trace' button loads a trace leading to the unsafe state in the editor, as seen in Figure 9b.


Simulation Trace

```
ProgCheck
(Check, G0, PreG1, Start, id54, Wait, __start
startrecov: Protocol → PatientToxic(0)
(FastRecover, Rec0, PreG1, Start, id54, Wai
g1: Toxic(0) → Patient
(FastRecover, Rec1, id39, Start, id54, Wait,
Toxic(0)
(FastRecover, Rec1, NNN, Start, id54, Wait,
Toxic(0) →
(FastRecover, Rec1, Recovered, Start, id54,
endrecov: Protocol → PatientToxic(0)
(id8, G0, id34, Start, id54, Wait, __start)
test: Protocol → Test(0)
(Check, G0, id34, id47, id54, Wait, __start)
Test(0) →
(Check, G0, id34, FalsePos, id54, Wait, __st
Test(0)
(Check, G0, id34, Start, id54, Wait, __start)
Toxic(0)
(Check, G0, Postrecov, Start, id54, Wait, __s
startrecov: Protocol → Patient
(FastRecover, Rec0, Postrecov, Start, id54,
endrecov: Protocol → Patient
(id8, G0, Postrecov, Start, id54, Wait, __star
```

(b) A partial view of the trace window in Uppaal. Selecting any of the states of this trace shows the exact state of the model at that time in the rest of the window.

Figure 9: Example usage of showing a trace to an unsafe state. This property configuration is part of the case study detailed in Section 8.1.1

names, edge segments or expressions as shown in the Uppaal editor view. The tool implements an extended `NSTA` to `Document` transformation with the original Uppaal model as additional parameter. The original model is used to set the positions of elements in the transformed model. As a result, the model shown in the trace view closely resembles the original model.

6 Reachability Algorithms

Some of the implemented validation properties can be generalised as either safety properties (an unsafe state is never reached), or reachability properties (a desired state is eventually reached). For greater flexibility in model validation methods, the tool separates *validation properties* from model checking procedures. A validation property is generalised to a procedure that performs some kind of model transformation, and returns a state formula specifying either a desired or unwanted state (i.e. a reachability property or a safety property).

In TCTL, safety properties can be expressed as $A\Box\neg\phi$, and reachability properties can be expressed as $E\Diamond\phi$. Since $A\Box\neg\phi \equiv \neg E\Diamond\phi$, we only require an algorithm that decides if an NHA M can reach a state satisfying ϕ.

The problem here is that reachability checking of hybrid automata is, in general, undecidable. However, for the purposes of model validation, an algorithm that is either unsound or incomplete could provide enough useful information to the modeller. Presented below are the different approaches to reachability checking that are used in this research.

6.1 Concrete Checking

The simplest way to perform reachability checking is to use a simulation-based approach. Here, the simulation algorithm produces n runs of the model, and terminates if at any time the state of the model satisfies ϕ, concluding that the state is reachable. However, if the algorithm has not found a run satisfying ϕ after n runs, no definitive conclusion can be made. This results indicates that either the state is unreachable, or that, by chance, the state has not been visited yet.

Even though the algorithm is not complete, it still provides useful information to the modeller. For large enough n, if no accepting run is found, the modeller gains confidence that the state is indeed unreachable.

Uppaal SMC provides a way to perform a number of simulations and filter the runs using a state formula. The syntax of this query is as follows:

```
simulate [<=t;n] {expression+} : m : φ
```

Where t is the global time bound of the simulation and n is the number of runs performed. The simulator plots the trajectories of one or more expressions specified in `expression+`. The runs are limited to the predicate ϕ, meaning that the simulation stops when ϕ is satisfied. If the run does not satisfy the predicate at any point it is discarded and not shown in the results. The simulator stops after m satisfying runs have been found.

We use the following query for reachability checking:

```
simulate [<=t;n] {0} : 1 : φ
```

Here, t and n are constants provided by the modeller, and ϕ is the state formula specified by the validation property. Because at least one expression must be provided, and we are not interested in plots of the simulation, we simply give the constant 0 as the expression list. After one satisfying run has been found it is known that ϕ is reachable, so the search can stop.

6.2 Symbolic Checking

Whereas the reachability problem is undecidable for hybrid automata, it *is* decidable for timed automata. Because TA are a sub-class of HA, it is possible to create a TA that is an abstraction of the original HA. Standard model checking algorithms can then be used to determine reachability. The limitation is that the abstract model is only an approximation of the original (concrete) model. This section first discusses an abstraction algorithm for hybrid automata, and the implication of model checking an abstraction instead of the concrete model. Next, the implementation of the abstraction algorithm for UPPAAL SMC models is discussed, along with other transformations required for practical model checking.

The terminology used in this section is derived from CEGAR [4] terminology. The original, untransformed model is called the *concrete* model, and is noted as M. An abstraction, or the *abstract* model of the original is noted as $\widehat{M}$. The *abstraction function h* maps a model to its abstract counterpart. The abstraction method presented here is derived from work on CEGAR techniques for timed automata by Dierks et al. [7]. Recall that in their work the basis of the abstraction function was variable hiding, where the abstract model only uses a subset of the set of clocks, and guards and invariants are relaxed so that they do not reference hidden clocks. The abstraction algorithm presented here is similar, but differs in that it is an HA to TA transformation instead of TA to TA.

6.2.1 Abstraction Algorithm for Hybrid Automata

If an HA contains a clock that can grow at a non-uniform rate (i.e. the derivative of the clock over time is not 1), reachability checking is undecidable in the general case [10]. The basis of our approach is to remove non-uniform clocks from the model while relaxing the guards and invariants as little as possible.

First, a set H containing all non-uniform clock variables is created. A non-uniform clock is a clock whose rate is defined and unequal to 1 in one or more invariants. All clocks in H are removed from the set of clocks in $\widehat{M}$. Next, all invariants, guards and updates are changed so that they no longer reference a variable in H. We define the abstraction function h over expressions as follows:

$$h(\phi_1 \wedge ... \wedge \phi_n) \equiv h'(\phi_1) \wedge ... \wedge h'(\phi_n)$$

$$h'(\phi) \equiv \begin{cases} \text{true} & \text{if } \phi \in E_H \\ \phi & \text{otherwise} \end{cases}$$

Where E_H is the set of all expressions containing a variable in H. The expression abstraction h is chosen such that if ϕ is satisfied for some valuation v, $h(\phi)$ is also satisfied for v.

Lemma 1. *For any valuation v, $v \models \phi \Rightarrow v \models h(\phi)$.*

Proof. To prove Lemma 1, we assume $v \models \phi$ and attempt to prove $v \models h(\phi)$. For $\phi \equiv \phi_1 \wedge .. \wedge \phi_n$, since $v \models \phi$, we have $v \models \phi_i$ for $i \leq n$. For ϕ_i, we have either:

- $h(\phi_i) \equiv \phi_i$. In this case, $v \models \phi_i$ implies $v \models h(\phi_i)$.
- Or $h(\phi_i) \equiv$ true. Trivially, in this case $v \models h(\phi_i)$.

Since we have $v \models h(\phi_i)$ for $i \leq n$, and $h(\phi) = h(\phi_1) \wedge .. \wedge h(\phi_n)$, we have shown that $v \models \phi \Rightarrow v \models h(\phi)$.

□

Proof Sketch of Over-Approximation

A general contract for the abstraction function h is that the abstracted model $\widehat{M}$ must be a strict over-approximation of the concrete model M. Formally, we define this property as $s \rightarrow s' \Rightarrow h(s) \rightarrow h(s')$. This means that for any state, if M is able to take a transition, $\widehat{M}$ must be able to take the same transition. To prove this, we consider both kinds of transition that an NHA or NTA may take, delay- and action transitions.

For delay transitions, we have $(L, v) \xrightarrow{d} (L, v')$, where $d \in \mathbb{R}^+$, L is the location vector, and v and v' are valuations of the clocks and data variables. We have $v' = v + d$, where v' is obtained by delaying all clock valuations by d (according to clock rates). Delay transitions are restricted by invariants, where for all $d' \in [0, d]$, $v + d' \models Inv(L)$ must hold (where $Inv(L)$ is the conjunction of the invariants in L).

If we have $s \xrightarrow{d} s'$ in M, we must also have $h(s) \xrightarrow{d} h(s')$ in $\widehat{M}$. If $s \xrightarrow{d} s'$ is a valid transition, then the invariants hold for any delay up to d. Using Lemma 1, we know that the abstracted invariants also hold, and therefore the transition $h(s) \xrightarrow{d} h(s')$ is valid in $\widehat{M}$.

For action transitions, we can take a similar approach. An action transition is a transition $(L, v) \xrightarrow{c} (L', v')$, with c either a channel or τ (signifying an internal transition). Among other restrictions, the transition is enabled when v satisfies all guards of the involved edges, and v' satisfies the invariants in L'. Using Lemma 1, we know that if an action transition is enabled in M, then $h(v)$ satisfies the relevant guards in $\widehat{M}$, and $h(v')$ satisfies the abstract invariants. The only updates that are removed are those assigning values to non-uniform clocks, so the valuation of every 'uniform' clock is equal between v' and $h(v')$.

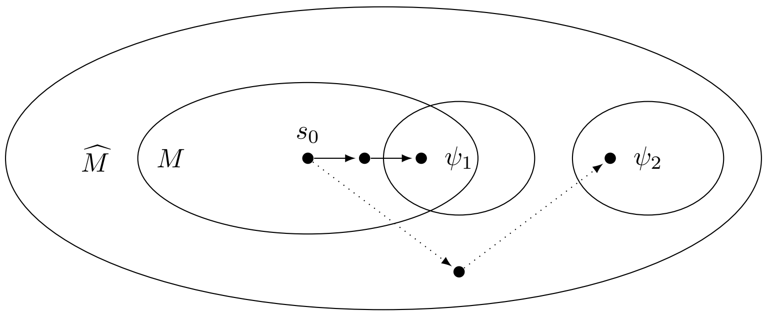


Figure 10: The reachable state space of a model M and its abstraction $\widehat{M}$, and state predicates ψ_1 and ψ_2.

Implications of Over-Abstractions

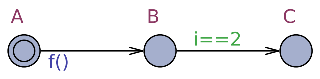
Since the abstract model $\widehat{M}$ is an over-abstraction of M, checking safety properties using this method is sound, but not complete. So for any safety property $\phi_s \equiv A\Box\psi$ with ψ a state predicate, $\widehat{M} \models \phi_s \Rightarrow M \models \phi_s$, but conversely $M \models \phi_s \not\Rightarrow \widehat{M} \models \phi_s$. Similarly, for reachability properties $\phi_r \equiv E\Diamond\psi$, property checking is complete since $M \models \phi_r \Rightarrow \widehat{M} \models \phi_r$, yet not sound.

These implications are illustrated in Figure 10, where the regions marked M and $\widehat{M}$ note the set of reachable states of a model and its abstraction respectively, s_0 being the initial state, and ψ_1 and ψ_2 note the sets of states that satisfy the two state predicates. Here, checking the formula $E\Diamond\psi_1$ using an over-approximative model would result in the correct answer, seeing as a state in ψ_1 is indeed reachable from s_0 (as shown using the solid transition sequence). Using the same method to check $E\Diamond\psi_2$ results in a false positive, since it is only reachable through states in $\widehat{M}$ and not M (shown by the dotted transition sequence).

6.2.2 Abstractions of UPPAAL SMC Models

We now describe how the abstraction method for NHAs is applied to Uppaal SMC models. Since the Uppaal and Uppaal SMC formalisms are extensions to NTAs and NHAs additional transformations are required. Specifically, Uppaal SMC supports models with `double`-type floating-point variables, which cannot be used with symbolic model checking. The abstraction algorithm is extended so that the set H of hidden variables not only contains clocks with a non-uniform rate, but also all data variables of type `double`.

This causes complications because the model may contain functions whose control flow is dependant on a hidden variable. Consider the model in Figure 11, where a function may assign different values to a variable depending on a condition expression containing a `double`-typed variable. If the abstraction algorithm were to simply remove the reference to `d` in the branch condition, changing `d`



```
double d;
int i;

void f() {
    if (d < 0) {
        i = 1;
    } else {
        i = 2;
    }
}
```

Figure 11: An Uppaal SMC model with control flow dependant on a floating-point variable.

`< 0` to `true`, the resulting model would not be a strict over-approximation of the concrete model. In this case, the `A` $\rightarrow$ `B` transition would always result in $i = 1$, making location `C` unreachable. Since location `C` can be reachable in the concrete model, this abstraction method does not always result in an abstract model that is an over-approximation.

The abstraction method is extended to also hide functions, as well as clock and data variables. To prevent the above problem, the implemented abstraction method also hides all variables that may be written to by a hidden function. The abstraction is defined so that none of the data variables in the model are affected by the values of hidden variables.

The method has a black-box view of functions, only considering the set of variables that a function references, and the set of variables whose value may be updated. The algorithm creates a set of hidden variables H_V, and a set of hidden functions H_F. H_V and H_F are defined as the minimal set of variables and functions in a model that conform to the following rules:

1. All clock variables with a non-uniform rate and all data variables of type `double` are in H_V.

2. If a function contains any identifier expression of a variable in H_V, that function is in H_F.

3. All variables on the left-hand side of an assignment expression [3] in a function in H_F are in H_V.

4. All functions that appear in a function call expression in a function in H_F are also in H_F.

[3]An assignment expression being any expression that may change the value of a variable. In Uppaal this is either the statement `v=e` with `e` an expression (or `v+=e`, `v*=e` etc.), or a post-/pre- increment/decrement expression (`v++`, `v--`, `++v` or `--v`).

5. All functions that contain a function call expression to a function in H_F are in H_F.

Rule 2 states that any function that has any reference to a hidden variable is also hidden. Since the control flow of a hidden function depends on hidden variables, any variable that the function may write to must also be hidden, resulting in rule 3. If a hidden function `f` calls another function `g`, the variables written to by `g` are affected by the hidden function `f` and must therefore also be hidden, resulting in rule 4. Similarly, if a function calls a hidden function, its control flow is dependant on a hidden function and thus must be hidden as well, resulting in rule 5.

The algorithm is implemented using mutable sets H_V, initially filled according to rule 1, and H_F, initially empty. For every function in the model, the algorithm iterates over the nodes of its AST, adding any variable or function that conform to rules 2 to 5 to the sets. This process repeats until no new variable or function is added to H_V or H_F.

It can be seen that the proposed abstraction algorithm is coarse, and produces an abstract model which may hide more behaviour than is strictly required. Future work may research alternate approaches in an effort to produce finer abstractions (i.e. abstractions where fewer variables are hidden). This could be performed by analysis of the control-flow graph of the function, to determine whether the values of data variables are affected by expressions over hidden (`double`) variables.

The implemented abstraction algorithm supports a limited but significant subset of Uppaal SMC models. One of the unsupported features of Uppaal SMC is the dynamic creation of processes, where during runtime process can 'spawn' an arbitrary amount of new processes to be added to the model.

6.2.3 Time Bounding

With Uppaal SMC, every query is *bounded*, meaning that the SMC algorithm generates runs that end when the value of a clock x exceeds d. When no clock is specified, x is used to signify global time, and is substituted for a new clock that is never reset. The modeller might therefore rely on the fact that simulations of the model will eventually terminate. This, however, presents a problem when using our TA abstraction method, as TCTL formulas are not bounded. For example, a cycle-based Uppaal SMC model might contain some counter variable that keeps track of how often a certain event occurs. Since simulations of the model are bounded, the value of this variable will never exceed some finite value. If this model is subsequently abstracted to a TA and model checking is performed, there is no end state of the system. The value of the counter variable would thus grow indefinitely, making the state space of the model infinite.

Our tool provides a practical solution to this problem. When using TA abstraction, the user has the option to specify an optional clock x and a time-bound for said clock. If no clock is specified, x is assigned to a new global clock that is never reset. When this option is enabled, an additional automaton is

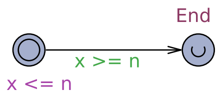
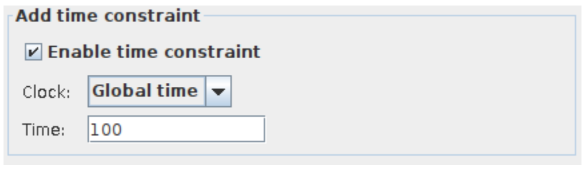


Figure 12: A time-bounded automaton.

added to the network, as visualised in Figure 12. When $x = n$, an invariant restricts further delay transitions and the automaton enters an urgent location `End`. In this state, the model is not able to take delay transitions, but only action transitions, ensuring that a state where $x > n$ is unreachable.


Add time constraint

☑ Enable time constraint

Clock: Global time

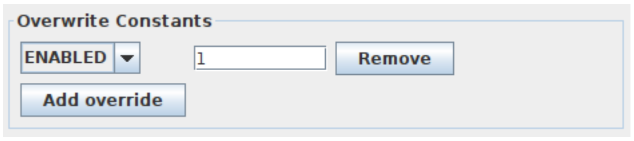
Time: 100

Figure 13: The time bounding dialog in the Uppaal plug-in window.

6.2.4 Constant Variable Overwrite

As this method uses model checking methods over the abstract model, it is subject to the problem of state-space explosion. Uppaal SMC queries are simulation-based and therefore do not suffer from the same problem. As a consequence, the creator of an Uppaal SMC model does not have a large incentive to reduce the state space of said model. This presents an obstacle for symbolic checking of NHAs, as the abstract model might have a state space too large to be checked in a practical amount of time. If this is the case, our tool offers some simple methods to simplify the model enough to make model checking feasible.

Uppaal models often contain multiple similar processes that are instantiated from one template (the train-gate model is an example of this). The amount of template instantiations is often determined by some constant N in the global declarations of the model. The complex behaviour that occurs when a number of processes interact is often of interest to the modeller. Every additional process increases the symbolic state-space exponentially. However, for the purposes of model validation, one or two instantiations might be enough to find errors or to verify correctness.


Overwrite Constants

ENABLED 1

Figure 14: The constant overwrite dialog in the Uppaal plug-in window.

To effectively allow the user to change the number of process instantiations when using symbolic model checking, a pre-processing step is added to the validation method. Figure 14 shows the UI dialog of this option in the UPPAAL editor. The user is able to specify zero or more global constants of the model, and assign a new value to them. This changed model is then used for the validation procedure.

7 Validation Properties

This section discusses the definitions and implementations of the validation properties used in the model validation tool. Sections 7.1 through 7.3 discuss the original properties developed for this work, and Section 7.4 describes how a subset of the checks in UrPal were implemented in the tool.

7.1 Model Invariant

The simplest and most general validation property implemented is the model invariant. This property states that all reachable states of the checked model should satisfy the given invariant ϕ. Expressed in TCTL, the property is defined as $M \models A\Box\phi$, with M the model under test. The invariant is the only parameter of this property, and can be noted as any valid UPPAAL state predicate. This makes it possible to express properties over the variables of the model (e.g. $x > 4$) or the currently active location in a process (using the `Process.Location` syntax). Additionally, the `deadlock` keyword can be used if the property is verified using symbolic model checking (deadlock checking cannot be performed using UPPAAL SMC queries because it relies on finite runs of the model).

The modeller can use this property to ensure that states that should be unreachable if the model is correct are indeed not reachable. If a state that does not satisfy the invariant *can* be reached, the model is considered invalid. Depending on the reachability checking method used, either a symbolic or a concrete trace to such a state can be shown in the UPPAAL editor. The trace shows a transition sequence leading to the incorrect state, which can help the user to determine the exact cause of the incorrect behaviour and resolve the error.

7.2 Synchronisation Check

It was observed that a common pattern in UPPAAL models is to have a process that should be able to react to a synchronisation in every possible state. For example, if a different process sends a reset signal, the receiving process should always return to an initial location and state. Because the modeller has to manually add the receiving transitions, it is not unlikely for a mistake to occur, which would lead to incorrect behaviour of the model. In UPPAAL SMC, this can occur even more frequently, because only broadcast synchronisations are allowed. Broadcast synchronisations are one-to-many, and are non-blocking, meaning that the sending process can always take a synchronisation-sending transition, even if there is no process ready to receive the synchronisation.

In order to find mistakes related to failing to synchronise, we added a mechanism to check whether a process in an NHA is always capable of synchronisation on a given channel. Expressed more formally, the checked property is satisfied if all processes P instantiated from a given template T take a receiving-edge transition when some other process takes a sending-edge transition, unless the ignore condition ϕ is satisfied. The modeller can specify an ignore condition to filter out false-negatives. For instance, a process would probably not have to react to a reset signal if it already is in the initial state.

7.2.1 Implementation

Our approach works by transforming the model to count the number of receiving-edge transitions taken, and checking if it equals the expected value. Given a user-specified channel c and template T, let n be the number of processes instantiated from this template. The model is extended with a global integer variable m, representing the number of receiving-edge transitions 'missed'. On every edge in the model with the synchronisation $c!$, the update $m := n$ is appended. On every edge of template T with the synchronisation $c?$ or $c!$, the update $m := m - 1$ is appended. Then, the property $m \neq 0$ is checked for reachability. If it is reachable, a process is unable to take the receiving-edge transition in some state, and an error is reported to the user.

The above method is able to work because in Uppaal, updates are evaluated sequentially, not concurrently. Furthermore, updates on the sending edge are evaluated before updates on the receiving edge. This means that setting the value of m to n always happens before the decrements. So, if all n receiving processes are able to synchronise, m is decremented n times, and equals 0 again after the transition. The decrement update is also added to a sending edge if it belongs to template T, so that the check is still correct if the sending process is one of the processes instantiated from T. This update must be evaluated after the initial assignment of m to n.

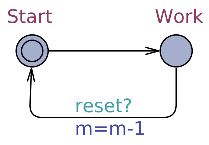
However, incorporating an ignore condition ϕ requires additional model transformations. It is not sufficient to simply check reachability for the state $m \neq 0 \land \neg\phi$, because if a 'missed' synchronisation occurs while ϕ is true, the model remains in a state where $m > 0$. It is then possible that the model can reach a state where ϕ becomes true and the value of m remains unchanged, which would lead to a false positive reported to the user.

To illustrate this, consider the example in Figure 15. Here, the goal is to check if the `reset?` synchronisation is always taken in process P, unless the process is already in the initial location `Start`, so ignore condition $\phi \equiv P.Start$. With the model transformation as described above, the predicate $m \neq 0 \land \neg\phi$ is reachable with the following transition sequence:

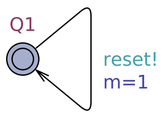
$$((Start, Q1), m = 0) \xrightarrow{reset} ((Start, Q1), m = 1) \rightarrow ((Work, Q1), m = 1)$$

It can be seen that this is a false positive, because the *reset* synchronisation was taken while process P was in the `Start` location, even though the ignore condition was meant to filter out this case.

To circumvent this incorrect behaviour, the model transformation must be extended so that the searched property can only be true after a synchronising



(a) Process P.



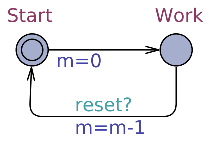
(b) Process Q.

Figure 15: An example of an NHA communicating with a broadcast channel.

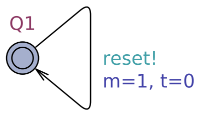
transition, and that any action or delay transition after that makes the property false. The model transformation is extended as follows:

- On every edge that does not synchronise over the specified channel c, the update $m := 0$ is appended.
- An additional clock t is added, and the update $t := 0$ is appended to every edge labelled with $c!$.

Now, the property $\psi \equiv m \neq 0 \wedge \neg\phi \wedge t = 0$ is checked for reachability. The first additional model transformation ensures that any action transition reset m, and thus the state $m > 0$ cannot persist between action transitions. The clock reset of t and the additional condition $t = 0$ ensures that any delay transition after $\xrightarrow{c}$ invalidates the property.



(a) Process P.



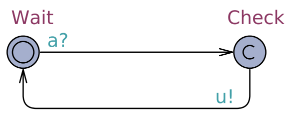
(b) Process Q.

Figure 16: The NHA of Figure 15 with model transformations supporting ignore conditions.

Figure 16 shows the NHA of Figure 15 with the extended model transformations. It can be seen that the transition sequence that led to the false positive earlier is now no longer possible, because the edge $Start \to Work$ now has the update $m := 0$.

7.2.2 Usage

The results of the reachability check of ψ are reported to the user. If ψ is unreachable, the model behaves according to the specification, and a positive result is reported. If not, Uppaal generates a trace (either concrete or symbolic), which is loaded in the editor for the modeller to inspect. The trace shows the exact transitions taken to reach the invalid state, and allows the modeller to deduce why the model was able to reach the undesired state.



```
broadcast chan a, u;
chan priority default < u;
```

Figure 17: The checker process for synchronisation post-condition verification.

7.3 Synchronisation Post-Condition

This property allows specifying an expression that should always be satisfied directly after a synchronisation on a broadcast channel takes place. Expressed formally, the property is valid if for a given state predicate ϕ and channel a, the model cannot reach a state s where $s' \xrightarrow{a} s$ and $s \not\models \phi$. Essentially, the property allows the modeller to specify a contract for channel synchronisations.

7.3.1 Implementation

Since the Uppaal query language provides no mechanism to specify a state after a channel synchronisation, model transformations are required to verify the property. Our approach works by adding a checker process to the model that listens to the specified broadcast channel a. Figure 17 shows this template.

The process is constructed such that when a synchronisation over channel a occurs, the only enabled transition sequence is the following:

$$((l_1, ..., l_{n-1}, Wait), v) \xrightarrow{a} ((l'_1, ..., l'_{n-1}, Check), v') \xrightarrow{u} ((l'_1, ..., l'_{n-1}, Wait), v')$$

This specifies that the locations of the other processes and the clock valuation cannot change when the system is in the `Check` location.

The process remains in the `Wait` location until an action transition with channel a is taken, and the process enters the `Check` location. The model is defined so that while the system is in the `Check` location, the only enabled transition is the internal transition to the `Wait` location. Because the `Check` location is marked as committed, no delay transitions can be taken, and only edge transitions starting from a committed location are enabled.

To avoid the possibility of another process taking a transition from a committed location, another restricting mechanism is required. Channel priorities are used to ensure that the `Check` $\rightarrow$ `Wait` transition is the only enabled transition. In the Uppaal formalism, a transition is enabled if no higher priority transition is enabled. Channel priorities are defined in the global declarations of an Uppaal model where, for example, the declaration `chan priority a < b` means that channel `b` has a higher priority than channel `a`.

The model transformation adds an auxiliary channel `u` to the model, and ensures that it has the highest priority. If no channel priority was earlier defined in the model, the statement `chan priority default < u;` is appended to the global declarations. If the model already contained a channel priority statement, `u` is appended at the end so that all other channels have a lower priority. The synchronisation `u!` is added to the `Check` $\rightarrow$ `Wait` edge, guaranteeing that its transition takes precedence over all other transitions.

Unfortunately, channel priorities are not supported in the verification of Uppaal SMC queries. When concrete reachability checking is used for verification, the channel `u` is left out of the transformed model. As a result, it becomes possible for a false positive to occur if another process takes a transition leaving a committed location while the checker process is in the `Check` location. Since Uppaal SMC provides no other mechanism to prioritise a transition, this false positive remains a possibility. As a work-around, the modeller can manually relax the post-condition predicate ϕ such that the erroneous state is filtered out.

7.3.2 Usage

Like the two validation properties discussed above, this property can be checked either symbolically or using concrete simulations. Again, a trace leading to the invalid state can be loaded in the editor to determine the cause of the incorrect behaviour.

7.4 UrPal Checks

Besides the validation properties described in the sections above, our tool also supports a number of the sanity checks developed for UrPal. Since the sanity checks are verified using symbolic model checking, the Uppaal SMC models are first abstracted to a 'pure' Uppaal model before further verification. Using this method, it takes little engineering effort to adapt the existing sanity checks to support SMC models, only requiring changes to handle the updated architecture and user interface changes. As a proof of concept, the model validation tool currently supports two of the sanity checks, further discussed below.

7.4.1 Unwanted Deadlocks

In a timed automaton, a deadlocked state is a state for which no action transitions are enabled in the time-abstract transition system (i.e. a state s where only delay transitions are enabled, and no state s' with an enabled action transition can be reached from s). Since in some models a deadlocked state is part of the intended behaviour of the model, UrPal defines a *wanted* deadlock as a deadlocked state where one or more locations in the location vector has no outgoing edges. The assumption is that these locations are intended to be deadlock locations, and a deadlock state containing one of these locations is therefore not erroneous. Consequently, an unwanted deadlock is defined as a deadlocked state where all of the locations in the location vector have one or more outgoing edges.

The implementation first creates the set of locations in the model with no outgoing edges. The following UPPAAL query is checked symbolically on the abstract model:

```
A[] deadlock imply (P₁.L₁ or ... or Pₙ.Lₙ)
```

With $\mathtt{P_i.L_i}$ a location without outgoing edges. Similar to the validation properties, if the property is satisfied, a positive result is reported to the user. If not, a negative result is displayed, and the user has the option to load a trace leading to the deadlock state in the UPPAAL editor.

Since this property is a safety condition and the abstracted model is an over-approximation, verification is sound but not complete. In other words, if the result of the verification is positive, it is guaranteed that the concrete model is unable to reach an unwanted deadlock state. If the outcome is negative either the concrete model may indeed reach an unwanted deadlock state, or this state is only reachable because of the relaxed guards and invariants of the abstract model. If so, the user can inspect the symbolic trace, and determine whether this transition sequence is indeed possible in the original model.

7.4.2 Template Location Reachability

This property is satisfied when every location of a template is reachable by one or more of the processes instantiated from said template. To avoid having to verify a reachability query for every location in the model, UrPals implementation uses `meta`-declared variables to mark 'visited' locations during a state space search. Using this method, the state space of the model only has to be searched once. Full details of the implementation of this check can be found in work by Onis [14]. If one or more locations were found to be unreachable, a negative result is reported to the user. These locations are marked red in the editor to communicate exactly which are unreachable.

Our work extends this procedure to work with UPPAAL SMC models by performing the above routine on an abstract over-approximation of the model. Because the check is a reachability property, verification is complete but not sound. Effectively, this means that no false positives can occur. However, false negatives can occur when the relaxation of guards and invariants cause locations that are unreachable in the concrete model to become reachable.

8 Evaluation

8.1 Case Studies

8.1.1 Cancer Immunotherapy Model

The first case study examines an Uppaal model used in research of cancer immunotherapy healthcare processes. Specifically, it models side effects of the therapy (immune-related adverse events, called toxics in the model), and studies the impact of test accuracy on the progression of these toxics. The model was constructed based on expert consultations and calibrated using empirical patient data.

The system models the healthcare process of one patient, who can potentially contract six different toxics, each with a grade of severity. According to a formalised protocol, these toxics are periodically monitored and tested, where a test has a certain chance of a false positive or false negative. Depending on test outcomes and previous recovery steps, the protocol dictates either a recovery phase or palliative care for the patient. The system has concurrent processes for the protocol, the patient, a periodic check for disease progression, six toxic processes with varying parameters, each with a corresponding test automaton, and a monitor automaton used for statistical queries.

Three variants of the model are used in the case study, each representing different points in the model development process. In the first model a bug is present, which is partially fixed in the second model, and fully removed in the third variant. We refer to these model variants with the label 1A, 1B and 1C. Consultation with the model developer allowed the exact cause of the fault to be known. As part of the study, we emulate the bug fixing procedure by creating a specification that 'captures' the bug, and thus allows the evaluation of the tool in bug finding and proving the absence of errors.

The bug was caused by a toxic process failing to synchronise correctly with the protocol process. Behaviour of a toxic process changes if the protocol is in a recovery phase, so broadcast channels are used to synchronise this state. Erroneous behaviour occurred because the toxic process was able to reach a state where a synchronisation over these channels was not enabled, leaving the system in an invalid state.

Validation specification Synchronisation of the recovery state is done using the `startrecov` and `endrecov` channels for entering and exiting the state respectively. The validation specification is configured to have 'Synchronisation Check' properties for these channels on the toxic template. In two locations of the template, `PreG0` and `G0` (indicating a toxic with grade 0, i.e. no toxic), behaviour of the process is not dependant on the recovery state, so no synchronisation is needed. To model this, the ignore condition `exists (n: id_t)`

#	**Property Type**	**Parameters**

1	Synchronisation Check	template = Toxic
		channel = `startrecov`

2	Synchronisation Check	template = Toxic
		channel = `endrecov`

3	Model Invariant	condition = `Protocol.Palliative == ProgCheck.Palliative`
4	Unwanted Deadlocks	
5	Template Location Reachability	

Table 1: Overview of validation properties used in case study 1.

`(Toxic(n).GO or Toxic(n).PreGO)` is configured for the properties. This expression is satisfied when one of the toxic processes is in the `PreGO`/`GO` location, avoiding any false negatives in the model validation.

Another property was configured in order to verify that two processes synchronise properly. The protocol process and disease progression process both have a state representing palliative care for the patient. Whenever one of these processes is in such a state, the other should be as well. Furthermore, if one of the processes is *not* in a palliative care state, the other must not be as well. Since both processes can independently enter this state, it is possible that an error in the model is made, causing the processes to be out of synchronisation. To verify the lack of this error, a model invariant with the condition `Protocol.Palliative == ProgCheck.Palliative` is configured.

Lastly, two sanity checks adapted from UrPal were added to the specification. These are the unwanted deadlock check and the template location reachability check. Both checks are only verifiable using symbolic checking of the abstract model. An overview and numbering of all properties is given in Table 1.

Verification Two configurations were investigated, one using symbolic checking and one using concrete checking. In order to make symbolic checking possible, some changes to the model and validation specification were necessary. Without any changes, the model contains six toxic and test processes. With a cut-off time of 600 seconds no property could be verified due to the large state space of the abstract model. The model was adapted to introduce a constant variable `ENABLED` specifying the number of toxic and test processes that should be instantiated. This allows the validation specification to include a ‘constant overwrite’ to lower the number of instantiations to 1 or 2, which allowed symbolic model checking in a practical amount of time due to the reduced state space. Note that reducing the number of template instantiations may alter the behaviour of the model, and therefore verification outcomes may not be representative for the unaltered model. In this case, there is no direct communication between the toxic and text processes, giving confidence that we can adequately verify the properties on the reduced system.

It was also necessary to introduce a time bound on the system, as detailed in Section 6.2.3. Without this time bound a discrete variable keeping track of the number of monitoring cycles is able to increment indefinitely. This variable is used as an index for an array of parameters, and increasing this variable out of the expected range of cycles would lead to an out-of-bounds error when indexing said array, terminating the verification. Introducing a time-bounding automaton with an upper bound of 66 time-units (being 11 6-week periods, which is the intended time bound of the model) made the state space finite and thus allowed symbolic model checking.

In order to verify the properties using concrete simulations the only configuration required is the number of runs and the time bound of the runs. In this case, 10,000 runs up to a time bound of 66 time-units were performed. This highlights a practical benefit of the verification method, as no significant time or expertise of the user is required to configure it.

Results As expected, the verification outcome of the 'Synchronisation Check' properties on the `startrecov` and `endrecov` channels showed that the process was able to reach a state with no receiving synchronisation transition enabled, meaning that the model variant 1A is invalid according to the specification. A trace leading to this state was able to be shown in the editor, allowing the user to quickly spot a location without the required synchronisation edges. This parallels the manual bug finding procedure used in the development of the model, where this problem was alleviated by adding synchronising self-loops to the location. This step was repeated in the case study, leading to model variant 1B, after which the validation specification was verified again.

With this step applied, there was still a possibility for erroneous behaviour. This could occur if the protocol exited the recovery state and then entered the state again without any delay transitions between the actions. If this happened while the toxic process was in a specific urgent location, it could not synchronise over the `startrecov` channel, causing further erroneous behaviour. Verification of the 'Synchronisation Check' property was able to confirm that this was possible, and provided a trace leading to the described state.

This second fault is a demonstration of the use of the automatic verification of model validation properties. Whereas manual inspection and manipulation of the model was eventually successful in finding and fixing the fault, consultation with the model developer did indicate that this was a time-consuming and difficult process due to the complex interactions required to reach the invalid state. With the use of our tool, the presence of the fault was able to be shown quickly, and using traces was a practical way of finding the exact cause of the problem.

The fault was resolved by changing a location in the toxic process to be marked as committed instead of urgent. With this change, creating model variant 1C, it was possible to verify that the synchronisations were able to be received in all reachable states of the model. Notably, this outcome was

	1A		**1B**		**1C**	
Property	**Satisfied**	**Time (s)**	**Satisfied**	**Time (s)**	**Satisfied**	**Time (s)**
1	No	1.430	No	2.082	Yes	26.323
2	No	0.551	No	0.557	Yes	26.883
3	Yes	29.628	Yes	54.617	Yes	23.906

Table 2: Verification outcomes and computation time for case study 1 using symbolic model checking.

	1A		**1B**		**1C**	
Property	**Satisfied**	**Time (s)**	**Satisfied**	**Time (s)**	**Satisfied**	**Time (s)**
1	No	1.743	No	3.298	Yes	8.664
2	No	0.387	No	0.661	Yes	7.608
3	Yes	7.201	Yes	6.562	Yes	7.369

Table 3: Verification outcomes and computation time for case study 1 using concrete model checking.

verifiable using both concrete and symbolic model checking, both having equal results and similar traces to invalid states.

With every iteration of the model the model invariant of property 3 was satisfied. Since it is a safety property, symbolic verification is sound. This allows the conclusion that, at least in the reduced model, states satisfying the expressions are indeed unreachable. The same outcome was achieved using the concrete checking method.

Property 4 was also satisfied in all model variants, meaning that no unwanted deadlocked state is reachable. Again, because the deadlock check is a safety property verification through symbolic model checking of the abstraction is sound. Verification of property 5 indicated that one location in the monitoring process was unreachable. Closer inspection of the model indicates that this location was only reachable as part of a debug mechanism which was disabled in the analysed models. Therefore, the unreachability of this location was part of the intended behaviour of the model.

Tables 2 and 3 show the outcomes and computation times for the verification using both verification methods. The benchmarks were run on a Intel Core i5-3230M CPU, using Uppaal version 1.23. It can be seen that computation time with symbolic checking is significantly increased if the property is satisfied. This can be attributed to the fact that safety properties are verified, and to show the absence of an invalid state the entire state space must be searched. The same observation applies to the concrete checking method, where all runs of the system (in this case, 10,000 runs) must be performed if the unsafe state is not found.

Using 50,000 simulations, the probability of a run of model variant 1B entering the unsafe state described by property 1 was determined to be $0.027\% \pm 0.015\%$ with a confidence of 95%. This highlights the classical downside of the monte carlo method for model checking, being that rare events are hard to verify. In this case the number of runs required to verify the property with a false positive chance of 5% is $\frac{ln(95\%)}{ln(1-0.027\%)} \approx 11,093$.

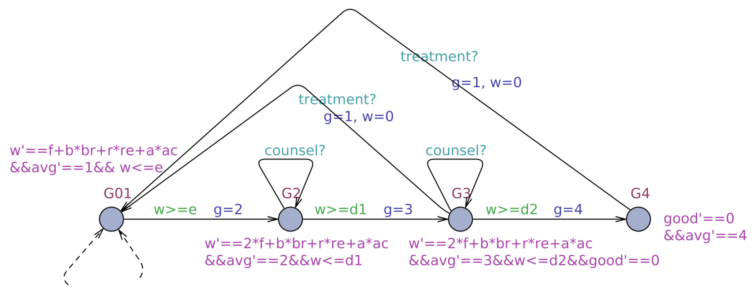


Figure 18: Partial view of the `Patient` template of the Tooth Wear Model.

8.1.2 Tooth Wear Model

The second case study uses an UPPAAL model developed for research of tooth wear monitoring [19]. It models a periodic counselling and treatment protocol over the entire life span of a patient. Analysis of the model was used to determine the benefits of a one-year monitoring cycle compared to a five-year cycle. Several aspects of the dental health of a patient such as enamel thickness and dietary influences on tooth wear are modelled using discrete variables. Depending on the stages of tooth wear, counselling or restorative treatment may be performed, each with a corresponding cost and effect on the tooth wear.

Compared to the first case study, the model contains fewer processes, only having a single instantiation of the three templates, being a process for the patient `Patient` (illustrated in Figure 18), the monitoring system `Twes` and a process marking the end of the simulation period `End`. Whereas in the first model non-uniform clocks were only used for monitoring costs, in the tooth wear model a non-uniform clock representing tooth wear *is* used in guards. This implies that in the abstract model used for symbolic model checking this clock must be hidden, and therefore the reachable state space is larger.

Validation Specification Cyclically, the `Twes` process checks the grade of the tooth wear and starts either a counsel or treatment. Synchronisation of this state with the `Patient` process happens using the `counsel` or `treatment` broadcast channel respectively. To validate whether the `Patient` process is able to take a receiving synchronising transition in all reachable states 'Synchronisation Check' properties are configured for the two channels.

After a treatment the tooth wear is restored, causing the patient to have a tooth wear classification score or 'grade' `g` of one. This behaviour is specified using a 'Synchronisation Post-condition' property, where the channel is `treatment` and the post-condition is `g == 1`.

The tooth wear grade is represented in two ways in the model: as one of four locations in the `Patient` process, and using a integer variable `g`. For correct behaviour of the model, it is important that these two mechanisms stay synchronised (i.e. if `g` equals 2, `Patient` is in the `G2` location). This is specified using a model invariant with the condition `Patient.G2 == (g == 2)`. Similar model

#	**Property Type**	**Parameters**

1	Synchronisation Check	template = Patient
		channel = `counsel`

2	Synchronisation Check	template = Patient
		channel = `treatment`

3	Synchronisation Post-condition	channel = `treatment`
		condition = `g == 1`

4	Model Invariant	condition = `Patient.G2 == (g == 2)`
5	Unwanted Deadlocks	
6	Template Location Reachability	

Table 4: Overview of validation properties used in case study 2.

invariants could be configured for the other possible grades, but are omitted for brevity.

Again, the unwanted deadlock and template location reachability checks were added to the specification, requiring no further configuration. All specified properties are enumerated in Table 4.

Verification Again, two configurations for the verification were made, one using symbolic model checking and one using concrete simulations. Similarly to the first case some additional configuration was required in order to be able to use symbolic model checking. A time bound of 74 time-units was introduced to limit the state space, being the age in years of the patient at the end of the modelled monitoring period.

In order to reduce the state space of the abstract model, and therefore reduce the verification time to a practical amount, additional work was required. In the original model enamel thickness was modelled using an integer variable. During the initialisation of the model, the value of this variable was randomly chosen from an uniform range using a select label. This select label picked an integer value l uniformly from the range $[0, 100]$. With SMC verification, this value is chosen stochastically. However, with symbolic verification this select causes 100 distinct transitions to be enabled, increasing the state space significantly.

In an effort to reduce the verification time the discretisation size was increased. This was done by replacing l with l' chosen with a select label with a range of $[0, 20]$. Every occurrence of l in the update statements was replaced with $l' \times 5$, ensuring that the same range of values could be chosen, but the amount of possible values was reduced with a factor 5.

For the verification using concrete model checking an upper bound of 10,000 runs was configured, with a time limit of 74 units. Again, no further configuration was required for the concrete method.

	Symbolic		**Concrete**	
Property	**Satisfied**	**Time (s)**	**Satisfied**	**Time (s)**
1	No	1.441	Yes	2.052
2	Yes	15.900	Yes	1.192
3	Yes	27.038	Yes	0.882
4	Yes	16.890	Yes	1.030
5	Yes	27.823	N/A	
6	Yes	16.892	N/A	

Table 5: Verification results and computation time for case study 2.

Results Table 5 summarises the verification outcomes for all validation properties using both verification methods, together with the computation time required to produce the outcome. Notably, all outcomes are in agreement with each other except for property 1. Recall that property 1 was a 'Synchronisation Check' property for the `counsel` channel.

The symbolic trace leading to the unsafe state of property 1 includes a transition of the `Patient` process from the `G3` to the `G4` location while the `Twes` process is in a location leading to a `counsel!` transition. Since counsels only happen with a wear grade of 2 or 3 the `Patient` process does not have a receiving synchronisation transition, leading to the invalid state. In the abstract model, the `G3` $\rightarrow$ `G4` transition is always enabled due to the hiding of tooth wear variable `w`, whereas in the concrete model this transition is only enabled when `w` exactly equals a constant. Further verification of the property using 500,000 runs indicates that either the unsafe state is unreachable in the concrete model or that reaching it is exceedingly improbable.

We conclude that the negative outcome of property 1 is a false negative with the current iteration of the model. It is however not unlikely for a change in the constant values of the model to enable the specific timing constraints leading to the error in the concrete model. A change in the marking of locations in `Twes` from urgent to committed would guarantee that the `Patient` process cannot take transitions before a `counsel` signal, alleviating the problem. Symbolic verification of the model with said change indicates that the property is valid, guaranteeing that the problem is no longer present.

8.2 Discussion

Based on the observations of the case studies, an evaluation of the implemented reachability algorithms and validation properties is performed. Each method is evaluated using the qualitative metrics detailed in Section 4.2.

8.2.1 Reachability Algorithms

Soundness and Completeness Table 6 provides an overview of the soundness and completeness of the reachability algorithms. Due to the over-approximative nature of the symbolic checking method, it is sound but not

	Safety Properties		Reachability Properties	
	Sound	Complete	Sound	Complete
Symbolic Abstraction	Yes	No	No	Yes
Concrete Simulations	No	No	Yes	No

Table 6: An overview of the soundness and completeness of the reachability algorithms.

complete for safety properties, and inversely, complete but not sound for reachability properties. The theoretical analysis of these statements is detailed in Section 6.2.1.

The concrete checking method is sound for reachability properties, since the presence of a trace leading to the specified state proves the property. However, for sufficiently rare events, the algorithm may not find a trace before the run limit is reached, so the method is not complete for reachability problems. For the same reason, the method is not sound for safety properties. A positive result indicates that either the specified state is unreachable, or that due to chance, the state was not reached by the run generation algorithm. As such, the method is also not complete for safety properties.

Time performance In both case studies it was necessary to reduce the state space of the models in order to reduce the computation times of the symbolic method to a practical amount. Even so, the symbolic method had higher computation times for nearly all cases compared to the concrete method. Only in cases where the state space did not have to be fully explored could the symbolic method perform roughly equally.

This results highlights the effectiveness of the statistical model checking method, since it can offer dramatically reduced computation times compared to symbolic model checking methods. The result can also be partially explained by the fact that the models were developed to be only analysed using Uppaal SMC queries. As such, no particular effort was made to reduce the state space, since it would not have a large impact on the computation times of statistical queries.

Utility Based on the results, it can be argued that the symbolic checking method is most effective in the verification of safety properties, since a positive result definitively proves that the property is valid. The concrete method cannot give the same guarantee for safety properties. During model development, the concrete checking method has distinct advantages in the reduced computation time, reduced configuration required, and lack of false negatives when verifying safety properties. Both methods proved capable of providing useful information to alleviate errors through providing traces to unsafe states.

8.2.2 Validation Properties

Here, we evaluate the three implemented validation properties used in this research. We refer the reader to [14] for an in-depth evaluation of the checks used in UrPal. The qualitative evaluation metrics are ease of use, improvement over existing methods, and applicability.

Model Invariant This is the simplest property to configure, requiring only one parameter, the safety condition. Since standard UPPAAL expression syntax is used, it should prove easy for most users to configure the property. Furthermore, the contract of the property (all reachable states must satisfy the expression) is simple.

The Model Invariant property could also be checked manually, without the use of our tool, using simple UPPAAL or UPPAAL SMC queries. The added benefit of the use of our tool to verify model invariants is that pre-processing steps such as constant overwrites or the TA abstraction method are automatically applied.

The property can be used as a general way to specify safety properties. Due to the expressiveness of UPPAAL predicate expressions, this property can be used in a wide range of use cases depending on the configuration.

Synchronisation Check This property requires more configuration, since the user has to specify a channel, a template and an optional ignore condition. Verifying this property using standard UPPAAL queries would require manual model transformations, which would be time-intensive and error-prone. The use of our tool improves on this by automating the process, eliminating potential mistakes.

This check can be applied in a large range of cases because of the fact that only broadcast synchronisations are allowed in UPPAAL SMC models. Because of this, a failure of two processes to synchronise does not result in blocking, and may therefore be challenging to detect. In these cases, this check is an effective way to verify validity.

Synchronisation Post-Condition This property also requires some configuration, since the user has to specify a channel and a post-condition expression. This expression can be seen as a specialised form of the model invariant, where the condition must hold in all states after a synchronisation, instead of in all reachable states. The check is useful because using UPPAAL state predicate expressions, it is not possible to specify a state after an action transition. To verify this property without the use of our tool would, again, require manual model transformations.

Like the model invariant property, the general form of the synchronisation post-condition property allows it to be used in a wide range of use cases. It can be used as a general way to specify safety properties related to process synchronisations.

9 Conclusion

In this work we examined methods for model validation of stochastic hybrid automata. The methods were implemented in a tool for Uppaal SMC models. Next, the tool was evaluated using practical case studies.

The case studies showed that the examined validation properties effectively allowed for the specification of a model. Using the tool, it is possible to create a formal specification which can be automatically verified. This is an improvement over the state of the art, since previously these properties could only be verified using manual model transformations. Use of our tool allows for an improved development process of models, where a formal specification is created and verified continuously. This work flow allows errors to be found early, and ensures that the model is valid throughout development.

As part of the research, a novel verification method using abstract over-approximations of hybrid automata was examined. The method uses a transformation from a hybrid automaton to a timed automaton to be able to use symbolic verification of properties. The case studies showed the effectiveness of the method, at the cost of requiring extra configuration to limit the state space of a model. It can be concluded from the case studies that the method is most effective when verifying safety properties, where a positive outcome definitively proves that unsafe states are unreachable.

Previously, no tools related to model validation of hybrid automata existed, and thus validation would have to be performed manually. We have improved on the state of the art by the introduction of a tool that provides an automated work flow to verify that the model under development is valid. Based on the case studies, we conclude that the examined methods provide an effective way of performing model validation.

9.1 Future Work

We consider the following research directions to be of interest for future work.

9.1.1 Hybrid Reachability Checking

With the symbolic and concrete reachability checking methods studied in this research both having specific advantages and disadvantages, the intuitive next step could be to combine the two methods. This 'hybrid' method could potentially share advantages of both methods while reducing the disadvantages.

A possible approach would be to adapt the procedure used to check if a trace is spurious, as detailed in work by Dierks et al. [7]. Here, it will be verified whether the trace obtained from the abstract over-approximation of the model is part of the language of the concrete model (i.e. check if the trace is spurious). This could be done using the same technique proposed by Dierks et al., in which a test automaton that only allows transitions from the trace is constructed. The product of the test automaton and the concrete model is then used for simulation-based verification. Because the traces obtained by symbolic model checking are themselves symbolic, i.e. describing sets of states transitioning to sets of states, a number of runs may be required to reach the specified state. If the symbolic trace is found to be spurious another trace may be computed and checking in the concrete model, which would be repeated until no unique traces remain.

The hybrid method has two main advantages. For one, if the abstract model cannot reach the specified state it is certain that it is unreachable. If not, the traces are used to 'guide' concrete runs, making it more likely that rare states are reached.

The proposed hybrid method was briefly explored as part of this research. However, assumptions of the UPPAAL run generation algorithm made it difficult to create a practical implementation of the method. This is because every process in a UPPAAL SMC model must not be time-locked, i.e. always able to take a delay or action transition. Because of this assumption it was not possible to construct a model transformation where exactly one process is able to take a transition in a given state according to the symbolic trace.

9.1.2 Improved Support for UrPal Checks

For this research, 2 out of the 8 sanity checks of the UrPal tool were adapted for UPPAAL SMC models. The checks were adapted so that the UPPAAL SMC model is first abstracted to a TA, after which the rest of the verification procedure is left unchanged. For extended functionality of our tool the rest of the sanity checks could be implemented using the same method. This is largely an engineering effort, requiring little research of new techniques.

However, verification using an abstract over-approximation can lead to false positives due to the fact that most sanity checks are reachability problems. A possible research direction could be to adapt the checks to use concrete simulations instead of symbolic model checking. A concrete verification method avoids some of the problems associated with symbolic checking, such as explosion of the abstract state space.

The concrete verification method of the checks could use a similar approach as the original method, where `meta` variables were used to store results between queries. Here, the SMC run generation algorithm is used to explore (most of) the state space, instead of the symbolic verifier.

References

[1] R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Information and computation*, 104(1):2–34, 1993.

[2] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.

[3] G. Behrmann, A. David, and K. G. Larsen. A tutorial on uppaal 4.0. *Department of computer science, Aalborg university*, 2006.

[4] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *International Conference on Computer Aided Verification*, pages 154–169. Springer, 2000.

[5] A. David, K. G. Larsen, A. Legay, M. Mikučionis, D. B. Poulsen, J. Van Vliet, and Z. Wang. Statistical model checking for networks of priced timed automata. In *International Conference on Formal Modeling and Analysis of Timed Systems*, pages 80–96. Springer, 2011.

[6] K. Degeling, S. Schivo, N. Mehra, H. Koffijberg, R. Langerak, J. S. de Bono, and M. J. IJzerman. Comparison of timed automata with discrete event simulation for modeling of biomarker-based treatment decisions: an illustration for metastatic castration-resistant prostate cancer. *Value in health*, 20(10):1411–1419, 2017.

[7] H. Dierks, S. Kupferschmid, and K. G. Larsen. Automatic abstraction refinement for timed automata. In *International Conference on Formal Modeling and Analysis of Timed Systems*, pages 114–129. Springer, 2007.

[8] T. A. Henzinger. The theory of hybrid automata. In *Verification of digital and hybrid systems*, pages 265–292. Springer, 2000.

[9] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. Hytech: A model checker for hybrid systems. In *International Conference on Computer Aided Verification*, pages 460–463. Springer, 1997.

[10] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata? In *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, pages 373–382, 1995.

[11] A. Hessel and P. Pettersson. Model-based testing of a wap gateway: an industrial case-study. In *International Workshop on Parallel and Distributed Methods in Verification*, pages 116–131. Springer, 2006.

[12] S. K. Jha, B. H. Krogh, J. E. Weimer, and E. M. Clarke. Reachability for linear hybrid automata using iterative relaxation abstraction. In *International Workshop on Hybrid Systems: Computation and Control*, pages 287–300. Springer, 2007.

[13] J. H. Kim, K. G. Larsen, B. Nielsen, M. Mikučionis, and P. Olsen. Formal analysis and testing of real-time automotive systems using uppaal tools. In *International Workshop on Formal Methods for Industrial Critical Systems*, pages 47–61. Springer, 2015.

[14] R. Onis. Does your model make sense? : Automatic verification of timed systems. Master's thesis, University of Twente, Formal Methods and Tools, December 2018.

[15] S. Schivo, B. M. Yildiz, E. Ruijters, C. Gerking, R. Kumar, S. Dziwok, A. Rensink, and M. Stoelinga. How to efficiently build a front-end tool for uppaal: a model-driven approach. In *International Symposium on Dependable Software Engineering: Theories, Tools, and Applications*, pages 319–336. Springer, 2017.

[16] U. Sørensen and C. Thrane. Slicing for uppaal. Master's thesis, Aalborg University. Department of Computer Science, 2007.

[17] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[18] F. Tip. *A survey of program slicing techniques*. Centrum voor Wiskunde en Informatica Amsterdam, 1994.

[19] P. Wetselaar, F. Lobbezoo, P. de Jong, U. Choudry, J. van Rooijen, and R. Langerak. A methodology for evaluating tooth wear monitoring using timed automata modelling. *Journal of oral rehabilitation*, 47(3):353–360, 2020.