Master's Thesis

Does your model make sense? Automated validation of timed automata

Ramon Onis

Supervisors: prof. dr. M.I.A. Stoelinga
prof.dr.ir. A. Rensink

University of Twente
Faculty of Electrical Engineering,
Mathematics and Computer Science
Formal Methods and Tools

December 7, 2018

Abstract

Timed automata are an important tool for modeling real-time systems. However, when timed systems become larger and more complex, modeling these systems becomes harder and error-prone. Making errors in such models and not detecting them might have serious consequences for the development of the system it resembles. Furthermore, the longer it takes to detect an error, the longer it takes to correct the error itself and reverse the consequences it might have caused. Therefore it is essential that the modeler is provided with ways to detect these mistakes in an easy way as early as possible. In this thesis, we present UrPal (*'your pal'*), a tool that performs sanity checks for commonly made errors when developing timed automata in Uppaal. For these common errors, efficient methods to detect them and present results to the user in a helpful manner have been designed and implemented. Our solutions makes extensive use of model-driven engineering, in particular model transformations. We show that the designed implementations are sound and correct and evaluate the performance in order to choose the most efficient implementations. Furthermore, we apply the sanity checker to several (industrial) models to show the value of the sanity checker.

Contents

List of Figures

List of Tables

1. Introduction

There are many systems in which time is a crucial aspect. For example, it is crucial for airbag systems in cars that they are always able to deploy on time. For such systems, it is crucial that it can be proven that they conform to these requirements. For airbag systems, it might be a critical requirement that the deployment time is within 30ms. Aside from real-world testing (e.g. dummy testing), formally proving the conformance of these systems to the requirements through modeling has been proven to be very effective and efficient in the past.

In order to model such systems, we must be able to express aspects as coordinated/concurrent behaviour and time. The world of formal methods provide a large set of formalisms that enable us to model many kinds of systems. After these systems are modeled, one could verify certain properties (e.g. what is the waiting time for a patient in a hospital?). One such formalism is timed automata (TA), which provide a useful framework for modeling such systems where time is important [1].

A single timed automaton might represent a train navigating through a railway network. By modeling multiple TAs and composing them in a system, more complicated systems with concurrent behavior can be modeled. This enables us to model multiple trains, railroad crossings and train stations, and how these components behave concurrently and coordinated. A useful property of timed automata is that it has been shown that the reachability problem of timed automata is *decidable* [1]. This makes it possible to formally verify the requirements of these systems. Thus, by correctly modeling an airbag using timed automata, we are able to prove through verification that a certain airbag system can deploy within the required time frame.

Because of the expressivity and verification possibilites of timed automata, several tools have been developed to model and verify them. UPPAAL is such a prominent tool suite that enables users to create and analyse networks of TAs [2]. Via a graphical user interface (GUI), (parametric) templates for TAs can be designed. By composing multiple templates in a system, a network of TA is made. This network can be simulated and, more importantly, be verified for a wide array of properties (such as reachability, deadlock, timing properties).

However, as the systems and their corresponding models get more complicated, it becomes more probable that the modeler will make errors. For smaller models, finding and correcting these errors is not a difficult task. However, for larger and more complicated models, this process of debugging might become overly complicated. Since the outcome of the verification crucially depends on the quality of the model, wrongly modeled systems might cause the outcome of verification to be inaccurate. Consequences for wrongly modeled systems can be severe. If, due to a human error, a model of an airbag skips an delay-introducing step, verification of that system might conclude that the reaction time is much smaller than in reality. This lets the manufacturer falsely believe that the airbag is safe, and thus approve it for production. A sanity checker can detect that certain locations and edges, that represent the skipped step, could not be reached. This would allow for the modeler to detect the error early on, and correct it.

In other areas (e.g. programming languages), static checkers that provide sanity checks have been long integrated in the workflow of a programmer in order to find errors. Static checkers, called linters, are an essential asset in the development environments of programmers [3]. More advanced tools have been developed to detect memory leaks or null-pointers is languages like C.

This thesis presents a tool that provides sanity checks for commonly made human errors in the modeling process of timed automata in Uppaal.
Through model transformations and Uppaal-queries, we implement the individual sanity checks as sound, complete, efficient and effective as possible. Queries will be at the base of running sanity checks, as we can then make use of Uppaal being able to efficiently verify them. Model transformations are used when queries alone are not sufficient to solve the sanity checks.
To make the tool helpful to the user, the outcome of the sanity checks gets presented to the user in a clear visual way. Using Uppaal's plugin system, the tool will be integrated in the GUI of Uppaal. This way, minimal effort is required by the user to integrate the sanity checker into the workflow of modeling systems in Uppaal.
Finally, performance testing show us which implementation for a certain sanity check is the fastest. Also, we show that the sanity checker is able to verify the integrity of a model by applying the tool to several (industrial) models.

2. Problem statement & research questions

Before defining this thesis's research questions, we will look at the actual problem statement which we have address. Based on this problem statement we formalize a broad research question that serve as the backbone of this thesis. To solve this question, we divide it up into several subquestions that have to be addressed in order to solve the main research question.

2.1. Problem statement

Ideally, a UPPAAL user should be able to detect mistakes easily and early on. Like in several other development environments, the user should be alerted if common mistakes are made in a model.
However, **in reality**, only basic syntax and compiler errors are presented to the user. Other mistakes that are easily detectable, like unwanted deadlocks or unreachable edges/locations, stay undetected to the user.
Consequences are that mistakes may not be detected at all, or only be detected when the model shows wrong behavior. If the mistake would not be detected at all, it could have severe consequences for the model. Erroneous models might cause wrong verification results, which can have negative consequences for the real-world system it represents (e.g. airbags that inflate too slow). Even if mistakes are detected after some time, correcting them might be problematic if this requires redesigning (part of) the model.
The solution to this problem would be to make a tool (sanity checker) that detects these commonly made errors for the user, and presents them in a effective way.

2.2. Research questions

Based on the problem statement, we address the following research question in order to ensure the sanity checker to be as helpful as possible to the user. With helpful we mean that the sanity checker enables the user to timely detect and correct common human errors.

> **Research question** How can a tool use sanity checks to help UPPAAL users find and correct command made modeling errors.

Thus, the main goal of this project is to design and implement such a tool.
To develop such a tool, we identify and formalize commonly made errors by UPPAAL users, develop ways to detect these errors in UPPAAL networks, develop ways of presenting the errors to the user, and finally evaluate the (time/memory/scalability) performance and effectiveness of the tool. These steps translate to the subquestions as shown below. Note that each subquestion contributes to the *helpfulness*

> **Subquestion 1** What are commonly made errors by users developing UPPAAL networks.

To answer this question, we collect commonly made errors from (experienced) UPPAAL. Also, we look at sanity checks that are already made (possibly in other tools or application areas).
Based on the findings, we select a list of common errors for which we will (attempt) to develop sanity checks. Also, the errors need to be formalized in terms of the actual syntax/semantics of networks of timed automata.

> **Subquestion 2** How can the selected errors be detected in a sound, complete, efficient and effective way.

To address this question, we present sanity checks that conform, as much as possible, to the following qualities:

- **Soundness:** this dictates that all detections are actual errors, and no false positives. Frequent false positives might cause the user to ignore the error.
- **Completeness:** this dictates that all errors are actually detected, such that there are no false negative. False negatives are obviously unwanted as one would want a sanity check to detect all errors.

- **Efficiency:** time and memory efficiency are needed to prevent the user to be discouraged to use the sanity checks (one would rather continue modeling that having to wait an hour for a sanity check). Also, very fast sanity checks enable on-the-fly checks, which will detect errors immediately after they are made.

- **Effectiveness:** Effectiveness means that the output of the sanity check actually contains enough information in order to identify the error (presenting the output to the user is the next step).

For more complicated models, we must accept that there is no *perfect* sanity check in terms of the above qualities. If, due to UPPAAL's approximation techniques, certain reachability problems are undecidable, we have to accept the lack of completeness and/or soundness. Efficiency and effectiveness are also qualities that can not both be achieved for certain sanity checks.

Subquestion 3 How can detected errors be properly communicated to the user

Now that the more theoretical part has been done. The errors are presented the users in such a way that:

- the user can understand what the error is based on the information given,

- the user can understand what has to be done in order to address the error.

As we are able to access the loaded model in UPPAAL, we use coloring to indicate reachability. Other errors that do not belong to a certain part of a model, are presented in a separate tab. We also use diagnostic traces that allow the user to inspect the details of the error.

3. Preliminaries

3.1. Timed Automata

Timed automata have been proven to be very useful in many applications where time has a crucial role. The ability to express a wide array of real-time systems and to formaly verify the requirements of these systems using specialized tools make them invaluable in many industries.
In this section we will present the reasoning, definitions, semantics of (networks of) timed automata in UPPAAL. For more detailed definitions, see [4].

3.1.1. Reasoning

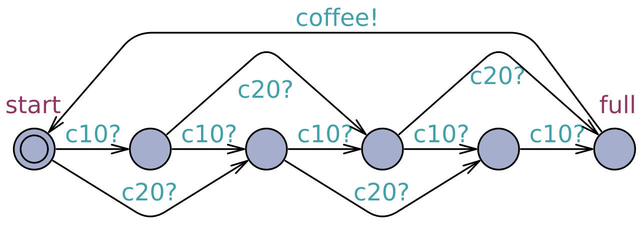


Figure 1.: A simple FSM representing a simple coffee vending machine

Timed automata are based on *finite state machines* (FSMs).
FSMs are a basic formalism which enables us to model the behavior of simple real-world systems. For example, we could model a coffee vending machine that serves coffee for 50 cents and only accepts coins of 10 and 20 cents as shown in Figure 1. The `start` locations is the initial location, the `full` location indicates that 50 cents have been inserted. The edge labels `c10?` and `c20?` models the action of a customer inserting 10 or 20 cents, while the label `coffee!` model the reaction of the machine to brew coffee.
However, the possibilities of this FSM are very limited as it only assumes the scenario in which a user buys a cup of coffee with only these types of coins.

Timed automata extend FSMs with clocks that can enable/disable edges.
On top of that, UPPAAL extends timed aumata by adding channels, which enable interaction between multiple parallel TAs, and data variables.
We will expand on each extension below.

Clocks FSMs don't provide a way to model time, apart from counting edges in a path. There is no way to express that the edge representing the insertion of a coin would take longer to the edge that represents the brewing of coffee.
Timed automata solve this problem by introducing real-valued clocks. With clocks, we can express the time it takes for the machine to brew a cup of coffee.
In Figure 2, we can see the basic mechanics of using clocks in timed automata.

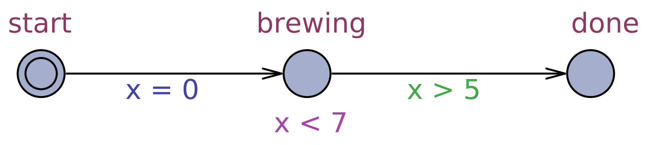


Figure 2.: A clock being used to express the time it takes to brew coffee

On the edge to the `brewing` location, we reset a clock to 0. While we are in this new location, an invariant specifies a condition which must hold while we are in this location. In this case, it only allows us to stay in this location while `x` is smaller than 7. The outgoing edge to the `done` location contains an guard. An edge with a guard is only enabled if the guard is true. In this case, the edge may only be taken if `x` is large than 5.
Combining the clock reset, invariant and guard results in that we must stay in the `brewing` location for at least 5 seconds, but no more than 7 seconds.

Concurrent behavior Now we have a way to express timing and delay, but note that while we are in a time-constrained location, no other actions can happen. In reality, it is possible for the user to insert coins or push on buttons.
We can solve this by separating tasks of a coffee machine into different timed automata: one for counting/refunding coins, one brewing coffee, one for handling pay-by-card and one that represents the actual customer. This way, each component only has to worry about it's own job.

Coordinated behavior Now that different parts of a coffee machine can behave concurrently, we still need to enable them to behave *coordinated*.
Uppaal extends timed automata with channels in order to support this. A channel has a sending and a receiving end. The sending end of the channel is indicated by appending the channel with an exclamation mark (e.g. `coffee!`) whereas the receiving end appends the channel with a question mark (e.g. `coffee?`).
When two automata have enabled edges, one sending to a channel and the other receiving a channel, the two automata traverse the edges simultaneously. This represents coordinated behaviour.
In Figure 3, we model a timed automata representing a person interacting with the

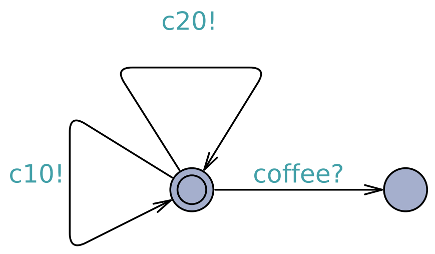


Figure 3.: A person that can put coins in a machine until it receives coffee

coffee machine in Figure 1. By sending to the channels `c10` and `c20`, the user inserts coins into the machine. This happens until the user *receives* coffee, which happens when the machine contains 50 cents.

Data variables Although we can now express time and coordinated/concurrent behavior, it is still hard to model the amount of cash that has been inserted into the machine. Problems arise when we want to take bigger prices into account of coins of smaller values. With the current expressivity, if we want to support 5 cent coins and a coffee thats worth 100 cents, we would need at least 21 states that represent having 0, 5,..., 195 or 200 cents in the machine.
Uppaal comes with data variables that can be written and read. This enables us to declare a integer variable representing the amount of cash that has been put in the machine.
The result can be seen in Figure 4, which has exactly the same semantics as the model in Figure 1. As the state of the machine is only determined by the amount of cash in it, we only need a single location. The two edges listing for the coin-insertion channels work as follows: listen for the coin-insertion channel (synchronization), check

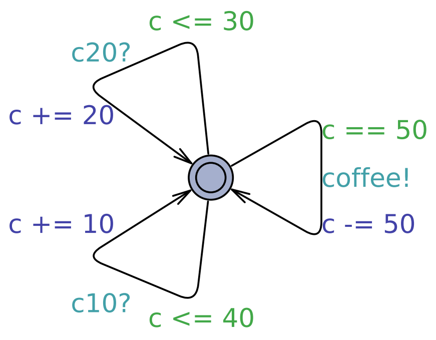


Figure 4.: The model in Figure 1, remade using data variables.

the coin doesn't cause too much cash (guard), and finally, if that condition is satisfied, increment the amount of cash (update). The edge that 'sends' a coffee works similarly: check the amount of cash (guard), send on the `coffee` channel (synchronization), decrease the cash to 0 (update).
We could easily extend the model further with data variables. For example by supporting more coins by store the types of coins in an array, supporting more types of drinks or enabling it to return change.

3.1.2. Syntax

We will now expand on the formal definitions of the aspects of timed automata.

Variables

Real-time systems require a framework that models time. We also require data variables to be modeled for many real-life systems (e.g. length of a queue, speed of a train, weight of a parcel). Also, a real-time system might enable/restrict certain actions based on these clocks/variables and update/reset them based on the action taken (e.g. disable a coffee machine if the coffee beans are depleted).
Data and time are expressed as clocks and data variables in Uppaal (collectively referred to as variables). Expressions consisting of these variables can be used to express guards of edges and invariants of locations. Also, variables can be updated on traversal of edges.

Definition 3.1 (Clocks and data variables). *Let $\mathcal{C}$ be a set of clocks that can range on $\mathbb{R}^+ \cup \{0\}$, and let $\mathcal{D}$ be a set of data variables that can range on discrete* *bounded domains. Let $\mathcal{V} = \mathcal{C} \cup \mathcal{D}$ be a set of variables consisting of clocks $\mathcal{C}$ and data variables $\mathcal{D}$.*

Definition 3.2 (Constraints). *Let $Const(\mathcal{V})$ be the set of constraints/boolean expressions over $\mathcal{V}$. We assume C-style operators for arithmetic, boolean logic, data comparison and array/member access.*

Definition 3.3 (Clock constraints). *Let $CConst(\mathcal{V})$ be the set of clock constraints, which are boolean expressions over the clocks $\mathcal{C} \subseteq \mathcal{V}$ in the form $x \bowtie e$ or $x - y \bowtie e$, where $x, y \in \mathcal{C}$, $\bowtie \in \{>, \geq, ==, \leq, <\}$ and $e \in \mathbb{N}$.*

Definition 3.4 (Guards). *Let $\mathcal{G}(\mathcal{V}) \subseteq Const(\mathcal{V})$ be the set of guards on $\mathcal{V}$, which consists of all conjunctions over $Const(\mathcal{D}) \cup CConst(\mathcal{C})$.*

Definition 3.5 (Invariants). *Let $\mathcal{I}(\mathcal{V}) \subseteq \mathcal{G}(\mathcal{V})$ be the set of invariants on $\mathcal{V}$, which consists of all guards that don't use lower bounds in clock constraints (i.e. $\bowtie \in \{==, \leq, <\}$ in Definition 3.3).*

Definition 3.6 (Updates). *Let $\mathcal{U}(\mathcal{V})$ be the set of updates on $\mathcal{V}$, which consists of sequences of assignments of the form $v := e$, where e is any expression if $v \in \mathcal{D}$ and $e \in \mathbb{N}$ if $v \in \mathcal{C}$.*

Example 3.1. *For the model in Figure 2, we would have $x \in \mathcal{C}$ as the time it takes to brew the coffee. The clock constraints $x < 7$ and $x > 5$ are used in the model. The constraint $x < 7$ is also an invariant.*

Channels

Most systems consist of multiple components behaving concurrently, but often not independently. It is therefore needed that multiple components can behave concurrently, while enabling interaction between them. Channels enable synchronization between TAs. The channels `c10` `c20` and `coffee` allows the TAs in Figures 3 and 4 to send signals between each other. When two TAs have edges enabled, one with a sending end of a channel (coffee!), and another with the corresponding receiving end (coffee?), both TAs can execute these edges simultaneously. A channel may be declared as a broadcast channel to enable the sender to synchronize with an arbitrary amount of receivers. If a channel is declared as urgent, then a synchronization transition on that channel must happen if it is enabled without any delay. Edges labeled with urgent channels are restricted from using clock constraints, as they would impose a delay on synchronization over an urgent channel, which is not allowed.

Definition 3.7 (Channels). *Let Ch be a set of channels, then $UCh \subseteq Ch$ and $BCh \subseteq Ch$ denote the urgent and broadcast channels resp.*

Definition 3.8 (Labels). *Let τ denote an internal unobservable action, then $Act = \{a!, a? \mid a \in Ch\} \cup \{\tau\}$ denotes the set of labels.*

Example 3.2. *When modeling a race, one might have multiple components representing contestants and a single race-controller.*
A channel `start` can be used to let the controller send a signal to all contestants to start. This channel must be declared as broadcast channel, otherwise, only one contestant will receive the signal.
A channel `finish` can be used by a contestant to indicate it has finished. This channel would be declared as urgent, otherwise a contestant with the `finish` channel enabled could experience a delay.

Locations

Locations can optionally be declared as urgent or committed. Urgent locations disallow delays if transitions are enabled, similarly to urgent channels. Committed locations, as an extension to urgent locations, are given a higher priority over non-committed locations; if a component is in a committed location, then components in non-committed are not allowed to execute transitions (unless this happens in synchronization with (an) other committed location(s)). We will regard committed locations as a subset of urgent locations, for sake of simplicity later on.

Definition 3.9 (Locations). *Let L be a set of locations. Then $ULocs \subseteq L$ and $CLocs \subseteq ULocs$ denote the sets of urgent and committed locations resp.*
Also, let $NULocs = L \setminus ULocs$ be the set of non-urgent (and therefore also non-committed) channels.

Timed Automata

We now present the definition of timed automata in UPPAAL. Note that some definitions disallow urgent and committed locations to have invariants as they can cause deadlocks. However, as the UPPAAL syntax DOES allow this, we will include this in our definition.

Definition 3.10 (Timed automaton(TA)). *Let a timed automaton A be defined by a tuple $(L, l_0, Lab, E, I, \mathcal{V})$ where:*

- *L is the set of locations*

- *$l_0 \in L$ is the initial location*
- *$Lab \subseteq Act$ is the set of labels*
- *$E \subseteq L \times Lab \times \mathcal{G}(\mathcal{V}) \times \mathcal{U}(\mathcal{V}) \times L$ is the set of edges. We denote the edge $(l, a, g, u, l') \in E$ with $l \xrightarrow{a,g,u} l'$*
- *$I : L \to \mathcal{I}(\mathcal{V})$ assigns invariants to locations.*
- *$\mathcal{V}$ the set of variables.*

Networks

We will now introduce networks, which consist of timed automata as components. With networks, we can express concurrent behavior in a system.

Definition 3.11 (Networks). *Let $A = \langle A_1, \dots, A_n \rangle$ denote a network of timed automata, were $A_i = (L_i, l_{0,i}, Lab_i, E_i, I_i, \mathcal{V}_i)$.*
Let $\mathcal{V} = \bigcup_{i=1}^{n} \mathcal{V}_i$ be the set of variables occurring in A, $I = \bigwedge_{i=1}^{n} I_i$ assigns invariants to location vectors A and $CLocs = \bigcup_{i=1}^{n} CLocs_i$ (where $CLocs_i \subseteq L_i$ is the set of committed locations in A_i) be all committed locations in A.

Definition 3.12 (Location vector). *A location vector of A is denoted as $\bar{l} = \langle l_1, \dots, l_n \rangle$, where $l_i \in L_i$. The notation $\bar{l}[l'_i/l_i]$ denotes the location vector that arises when l_i in $\bar{l}$ gets replaced by l'_i. Also, for any index set J, $\bar{l}[(l'_j/l_j)_{j \in J}]$ denotes the location vector that arises when l_j in $\bar{l}$ gets replaced by l'_j for each $j \in J$.*
$CLocs(\bar{l}) =$ denotes all committed locations in a location vector.
$\bar{l}_0 = \langle l_{1,0}, \dots, l_{n,0} \rangle$ denotes the initial location vector.

Definition 3.13 (Global/local variables). *The set of global variables, which are shared between two or more components, is defined by $\bigcup_{1 \le i \ne j \le} (\mathcal{V}_i \cap \mathcal{V}_j)$. All other variables are called local variables.*

3.1.3. Semantics

Now that we have defined the syntax of (networks of) timed automata, we will present the semantics.

Definition 3.14 (Valuations). *A valuation v maps clocks to non-negatives real values and data variables to their corresponding domains. Given a set of valuations $\mathcal{V}$, let $\mathbb{V}(\mathcal{V})$ be all possible valuations over $\mathcal{V}$. For an expression e, let $[\![e]\!]_v$ be the valuation of e under v.*

For any $v \in \mathcal{V}$ and $\delta \in \mathbb{R}^+$, we define the valuation $v + \delta \in \mathbb{V}(\mathcal{V})$ as follows: $\forall x \in \mathcal{C}.(v+\delta)(x) = v(x) + \delta$ and $\forall x \in \mathcal{D}.(v+\delta)(x) = v(x)$.
The initial valuation v_0 is defined as follows: $\forall var \in \mathcal{V}.v_0(var) = 0$.
$v \models g$, where $v \in \mathbb{V}(\mathcal{V})$.
$g \models Const(\mathcal{V})$ denotes constraint satisfiability.

Definition 3.15 (Updates on valuations). *Let any $u \in \mathcal{U}(\mathcal{V})$ be an update $var_1 := e_1, \ldots, var_m := e_m$ and $v_1 \in \mathbb{V}(\mathcal{V})$. Then let $v_{i+1} \in \mathcal{V}(var)$ with $1 \leq i \leq m$ be defined as follows: $v_{i+1}(var_i) = [\![e_i]\!]_{v_i}$ and $\forall var \in \mathcal{V} \setminus \{var_i\}.v_{i+1}(var) = v_i(var)$.*
Finally, we define $u(v_1) = v_{m+1}$ as the valuation after sequentially executing the assignments in update u on v_1.
Similarly, for an index-set $J = \{j_0, \ldots, j_m\}$, we denote u_J as the sequential execution of the updates $u_{j_0}, \ldots, u_{j_m}$.

The semantics of a network A can be described as a timed transition system $(S, s_0, \{\epsilon\} \cup \mathbb{R}^+, \rightarrow)$, where:

- $S \in L \times \mathbb{V}(\mathcal{V})$ is the set of reachable states. A state is denoted $s = \langle \bar{l}, v \rangle$.
- $s_0 = \langle \bar{l}_0, v_0 \rangle$ is the initial state.
- $\{\epsilon\} \cup \mathbb{R}^+$ is the set of labels in the transition system, where ϵ implies an action transition, and $\delta \in \mathbb{R}^+$ implies a delay.
- $\rightarrow \in S \times \{\epsilon\} \cup \mathbb{R}^+ \times S$ is the transition relation. We denote action/delay transitions as $s \overset{\epsilon}{\Rightarrow} s'$ or $s \overset{\delta}{\Rightarrow} s'$ resp., with $\delta \in \mathbb{R}^+$.

The transition relation is constructed as follows:

- Component-internal actions:
  $\langle \bar{l}, v \rangle \overset{\epsilon}{\Rightarrow} \langle \bar{l}[l'_i/l_i], u_i(v) \rangle$ for any edge $l_i \xrightarrow{\epsilon, g_i, u_i} l'_i \in T_i$ such that:

  $v \models g_i$, and

  $u_i(v) \models I[\bar{l}[l'_i/l_i]]$, and

  $l_i \in CLocs_i$ or $CLocs(\bar{l}) = \emptyset$

- Channel synchronizations:
  $\langle \bar{l}, v \rangle \overset{\epsilon}{\Rightarrow} \langle \bar{l}[l'_i/l_i, l'_j/l_j], u_j(u_i(v)) \rangle$ for any $l_i \xrightarrow{a!, g_i, u_i} l'_i \in T_i$ and $l_j \xrightarrow{a?, g_j, u_j} l'_j \in T_j$ such that:

  $a \notin BCh$, and

$v \models g_i$ and $v \models g_j$, and

$u_j(u_i(v)) \models I[\bar{l}[l'_i/l_i, l'_j/l_j]]$, and

$\{l_i, l_j\} \cap CLocs(\bar{l}) \neq \emptyset$ or $CLocs(\bar{l}) = \emptyset$

- Broadcast channel synchronizations:
  $\langle \bar{l}, v\rangle \overset{\epsilon}{\Rightarrow} \langle \bar{l}[l'_i/l_i, (l'_j/l_j)_{j\in J}], u_J(u_i(v))\rangle$ for any $l_i \xrightarrow{a!,g_i,u_i} l'_i$ such that:

  $a \in BCh$, and

  J is the maximal index-set such that for any $j \in J$, there exists an edge $l_j \xrightarrow{a?,g_j,u_j} l'_j \in T_j$, and

  $v \models g_i$ and $\forall j \in J. v \models g_j$, and

  $u_J(u_i(v)) \models I[\bar{l}[l'_i/l_i, (l'_j/l_j)_{j\in J}]]$, and

  $(\{l_i\} \cap \{l_j | j \in J\}) \cap CLocs(\bar{l}) \neq \emptyset$ or $CLocs(\bar{l}) = \emptyset$

- Delays:
  $\langle \bar{l}, v\rangle \overset{\delta}{\Rightarrow} \langle \bar{l}, v+\delta\rangle$, for any $\delta \in \mathbb{R}^+$ such that:

  $(v+\delta) \models I[\bar{l}]$, and

  $ULocs(\bar{l}) = \emptyset$, and

  no synchronization over urgent channels is possible from any state $\langle \bar{l}, v+\delta'\rangle$ where $\delta' < \delta$.

Definition 3.16 (Edge selection). *Consider the transition $t \in \rightarrow$.*
All edges that t is constructed from are called the selected edges of t.
We say that t selects $e \in E$ when e is one of the selected edges of t.

We regard a transitions *enabled* if it conforms to the above criteria, disregarding the criterion that the target state of action transitions can't violate the target locations. An enabled action transition whose target state would violate an invariant in it's location vector is special; by definition, it's target state is undefined, as a state which violates an invariant should not exist. Therefore it is not included in the transition relation of a network. However, we can still reason about these transitions.

3.2. UPPAAL

In this section we will explain UPPAAL as a graphical tool to develop networks of TAs using a simple example system as a running example.

The description of the system is that of a simple factory:

A factory consists of a single conveyor belt that spawns a predetermined finite array of items requiring either easy, average or hard work. Jobbers can take the next item on the conveyor and process them which might require tools. Easy jobs require no tools, average jobs require either a mallet or a hammer, and hard jobs require only a hammer. The time it takes to do a job depends on the difficulty and on the tool that is used: easy jobs take between 5 and 7 seconds, average jobs between 15 and 17 seconds with a mallet or between 10 and 12 seconds with an hammer, and hard jobs take between 20 and 22 seconds.

We will only consider the time it takes to do a job. While interesting, the time it takes to grab or return a tool or item, and the delay that the conveyor might cause are ignored.

We can identify three different types components in the system description: a conveyor, a tool and a jobber.

UPPAAL systems revolve around templates, which are named (optionally parameterized) TAs.

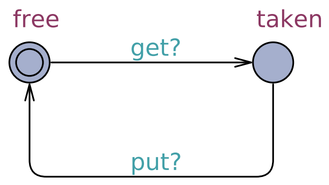


Figure 5.: The UPPAAL template for a tool

The most simple components, hammers and mallets, can be modeled by simple 2-location template consisting of one *free* location and one *taken* location (see Figure 5). We can immediately recognize the syntax of a TA in the graph: locations L are represented by nodes (where the initial location l_0 is represented by a doubly bordered node), the labels *get?*/*put?* attached to edges between the locations, implying the existence of the channels *get*/*put*.

Since a hammer and a mallet behave the same (i.e. they listen to a get and put channel), we can use parameters to instantiate both a hammer and a tool from the same template. By declaring the parameters `chan &get, chan &put`, we are able to instantiate a hammer and mallet with different channels:

```
hammer = Tool(get_hammer, put_hammer);
mallet = Tool(get_mallet, put_mallet);
```

Next, we can define a conveyor as a simple template, using 3 channels representing easy, average and hard jobs (see Figure 6). Note that we only name the start and end location, as only these are interesting for analysis.
For a possible extension to this template, we might choose to make the jobs randomly generated, or add time delays simulating a the movement of an actual conveyor.

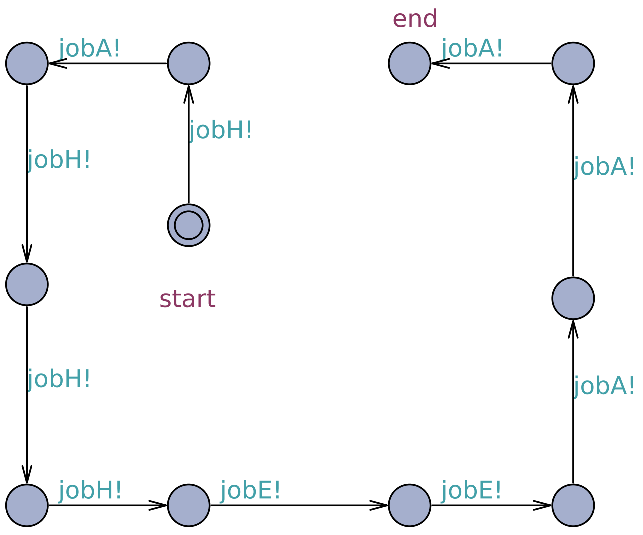


Figure 6.: The UPPAAL template for the conveyor belt

The most complicated component is the jobber. In order to model the jobber, we divide it's role into four separate tasks: taking an item, acquiring the necessary tool, executing the job, and returning the tool.
In the first step, we have three scenarios: getting a easy, average or hard job. This means means synchronizing with the conveyor by one of the channels `jobE`, `jobA` or `jobH`. Next, depending on the difficulty, the jobber now must acquire a tool: an easy job skips this step, an average has the option between a mallet and a hammer, and a hard jobs requires a hammer. Next we must model the passing of time depending on the difficulty and tool. Finally we return the tool. Acquiring and returning a tool uses channel synchronization similar to getting an item from the conveyor. To model the passing of time, we use a clock, and invariants and guards based on this clock: an invariant on a `work` location restricts the maximum job duration, a guard on the outgoing edge restricts the minimum job duration.
Translating this approach to an actual UPPAAL template results in the template as

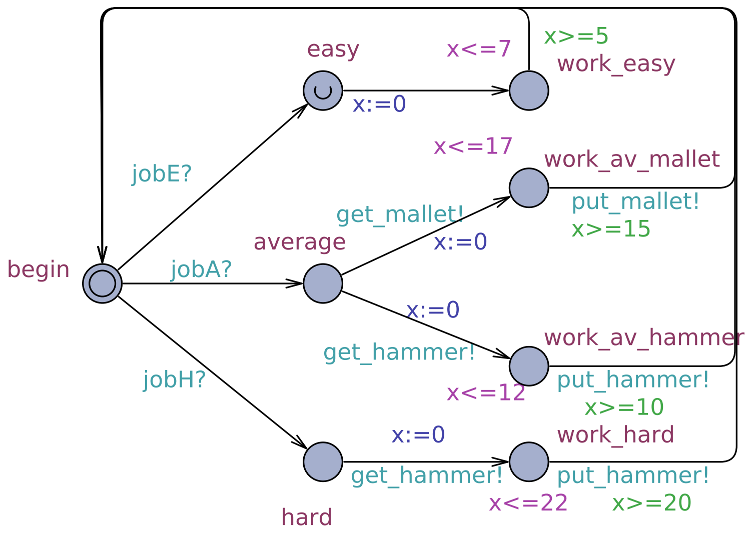


Figure 7.: The UPPAAL template for the jobber

shown in Figure 7. The four steps are clearly represented: the outgoing edges from the `begin` location simulate picking an item, the next edge acquires the tool, the `work` locations hold for some time, after which the tool is returned and the loop starts over again. Note that the edge between `easy` and `work_easy` can be omitted (i.e. the two locations can be merged into one) since no tools are to be acquired. However, in this case we must make the `easy` location urgent since acquiring tools for an easy task must not take any time.
Now that we have defined the templates, we can compose them into a system. Note that we can add as many hammers, mallets, conveyors and jobbers as we want. After that, UPPAAL enables the user to simulate the system by choosing every transition in sequence.
Additionally, UPPAAL supports writing queries in syntax based on computational tree logic (CTL) in order to validate certain properties. For example, if we want to know if it's possible for two jobbers to process the whole conveyor belt within 100 seconds, we could write the following query:

```
E<> (belt.end && jobber1.begin && jobber2.begin && now <= 100)
```

In plain English, this translates to: a state is reachable where the belt is in the `end` location, and the jobbers are in the `begin` location (i.e. they are done with their jobs) while the elapsed time `now` is smaller or equal to 100.

3.3. Uppaal architecture

There are many features that Uppaal adds to the original TA definitions by Alur & Dill. Apart from that, Uppaal provides concepts and tools that enable us to inspect the model itself, instead of the system it represents. Below, we will show the most important features that we use.

3.3.1. Meta variables

A crucial feature in Uppaal for our tool is the concept of meta variables. Meta variables.
When UPPAAL tries to check a query, it will execute a search algorithm on the state space of the system where a state consists of a location vector and a valuation [1] (see Definition 3.15). Meta variables are variables declared with the keyword `meta` (e.g. `meta int i`). Meta variables behave like normal variables, but are not considered part of the state: when two states only differ in meta variables, they are considered equal.
This feature is documented to be used for temporary variables so that they do not increase the state space [5]. One such example is when we want to swap the values of two variables:

```
int a = 3;
int b = 5;
meta int tmp = 0;
tmp = b;
b = a;
a = tmp;
```

In this case, `a` and `b` are part of the state as wanted. However, the value of `tmp` is not part of the state.
Another property of meta variables is that their valuations are preserved when the verifier's search algorithm backtracks or starts a new verification[2]. This makes them useful to track information about (multiple) verifications. E.g. the amount of transitions that were fired, or the locations that were visited.

[1]UPPAAL reduces the state space by, among other methods, merging sets of valuations in a single state using *regions* and *zones*. For simplicity's sake, we will disregard these optimizations.

[2]It is only documented that meta variables are not part of the state, however personal communication also revealed that their values are preserved in this way.

3.3.2. Model API

The GUI of UPPAAL, which is written in *Java*, uses a publicly available model API [6].
This API enables us to read and write the XML representation of a model to a Java representation (which is called a `Document`). However, this representation has limitations. Firstly, all elements (templates, locations etc.) in this `Document` are based on XML elements. Thus, the internal model is more of an augmented XML model than a more helpful domain model. Also, similarly to the XML representation of a UPPAAL model, the declarations language is still in plain-text.
Additionally, the API allows us to compile the `Document` into a `UppaalSystem`. This compiles the actual network of TAs that is given to the verifier of UPPAAL. This includes expanding the templates into processes (e.g. a single template can spawn multiple processes), and expanding select-edges (see UPPAAL documentation). However, this still doesn't compile the declaration language. This compiled system can be queried on programmatically (which will call the engine underwater) which can also provide traces if possible.

3.3.3. Plugin framework (beta)

In beta versions, UPPAAL includes a plugin framework that supports the development of loosely coupled plugins.
Without the need to access proprietary code of UPPAAL, plugins can live in a separate tab in the interface of UPPAAL in which it can communicate with the rest of UPPAAL.
Concrete features of this plugin framework are:

1. Read and write the current model that is loaded in the editor-tab. The model is read and writen in the form of a `Document` (see 3.3.2).
2. Spawn custom tabs in the UPPAAL GUI.
3. Read and write traces to/from the simulator (future work).

3.4. Model-driven engineering

Model-driven engineering is a software engineering methodology that focuses on using domain models as abstract representation of certain concepts. Domain models can be in graphical tools and requires no knowledge of specific programming languages. Code generation then makes it possible to automatically convert the domain models to language-specific representations (e.g. Java classes). This approach enables faster software development by reusability of models, improving compatibility between systems and by providing separation of concerns.

3.4.1. Metamodels

The details of a domain model, its structure and rules, are described in a metamodel. A metamodel, also called a model of a model, is at the base of model-driven engineering. While a model can be used to describe the properties of real world phenomena (e.g. the name of a person is 'John'), metamodels are used to describe the properties of the models themselves (e.g. a person has an attribute 'name').
Some frequently used applications of metamodeling are:

- Document Type Definition (DTD) files which are metamodels for XML files.
- XML Schema Definition (XSD) files which also serve as metamodels for XML files.
- JSON schemas, which has recently been developed for JSON[7].

3.4.2. Model transformations

A crucial aspect of meta models that is also used in our tool, is the ability to write model transformations. Model transformations allows one to transform a model conforming to one metamodel to a model conforming to another meta model. It is also possible to transform a model to a model conforming to the same metamodel, which is how our tool will apply model transformations. Model transformations can be written in special transformation languages, such as ATL [8], or in general purpose languages such as Java (which will be done in our tool).
In Figure 8, we present an overview for model transformations applied to metamodels. In the lower corners, we have M_a and M_b, which are the source and target models respectively. These models conform to their corresponding metamodels MM_a and MM_b. These metamodels conform to metametamodels, which is *Ecore* in our case. The transformation between the two models, M_t, could also conforms to a metamodel. This transformation metamodel can be a transformation language such as ATL. Even further, ATL is also expressed using the semantics of the *Ecore* metametamodel. In

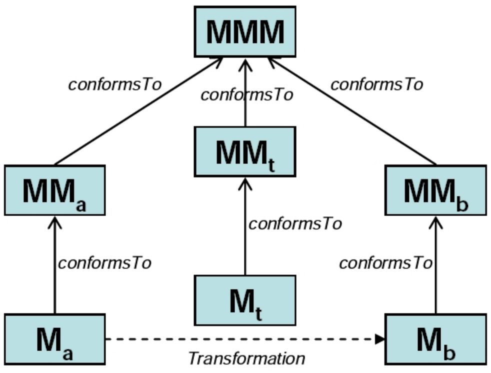


Figure 8.: Overview for model transformations

our case, we express model transformations programmatically in Java. Java could be seen as the metamodel for our transformations: MM_t.

3.4.3. Eclipse Modeling Framework

To facilitate model-driven engineering, the Eclipse Foundation has developed the Eclipse Modeling Framework (EMF). EMF provides a wide array of (runtime) tools aimed at model-driven engineering. At the heart of EMF lies Ecore, which is the core metamodel used in EMF.
Among others, EMF includes the language parser framework *Xtext* and the Epsilon Transformation Language (ETL) in which transformations between Ecore models can be developed.

As UPPAAL code base is closed-source, an open-source Ecore metamodel for UPPAAL models, queries and diagnostic traces has been developed [9]. This enables us to easily import, analyze, transform and generate UPPAAL models. All aspects of a UPPAAL model are included: the XML structure of origin UPPAAL files, along with the C-style declarations in which functions, variables, and systems can be declared. The Ecore *package* of UPPAAL templates can be seen in 9. At the middle of this model is a Template, which contains edges and locations (of which there is one initial location). Locations can contain invariants and a time kind (normal/committed/urgent). Edges contain guards and updates (of which there can be multiple). Note that this conforms to the definitions of timed automata in 3.1. Furthermore, edges might contain a synchronization on a channel on either the receiving of sending end. Also, an

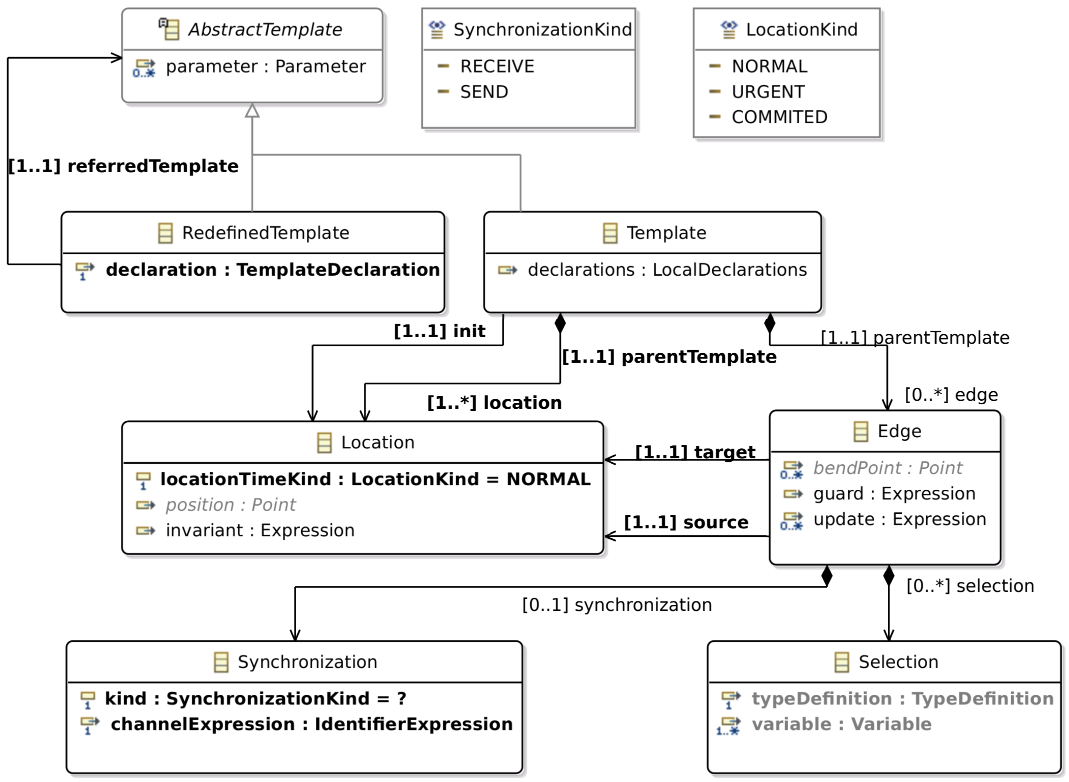


Figure 9.: Ecore model for templates

edge might contain a selection, which is an UPPAAL feature that non-deterministically binds a variable to a value in a given range. To bind parameters in templates we can use a RedefinedTemplate, which contains a TemplateDeclaration that contains the parameter binding. TemplateDeclarations, and other declarations, are part of another *package*. Other packages include other aspects of a UPPAAL model, such as expressions, statements or types.

3.5. Conclusion

We have now covered all prerequisite knowledge that covers the fundamentals for building the sanity checker.
The reasoning, syntax and semantics behind timed automata have been discussed and formalized. In addition we have covered the additions of UPPAAL to the framework of timed automata: data variables, channels and, most importantly, meta variables. We have also covered the plugin framework that allowed us to make the sanity checker a built-in tool inside the UPPAAL GUI, and the model API provided by UPPAAL that allows the sanity checker to access, change and verify models that have been loaded in the editor.
Finally, we have presented model-driven engineering as a valuable tool for transforming models.

4. Related work

In this section we will look at the existing applications of sanity checks.
First we will look at a very specific sanity check for zeno runs in UPPAAL models. This will provide examples of methods we can use to develop our sanity checks. As this sanity check is also complicated in terms of performance, we can see what ways exist to improve time/space performance of the sanity checks.
Next we will look at two different framework for (timed) systems, one for which a large array of sanity checks exist, while the other serves as a DSL for banking products. This will give us inspirations for sanity checks we could choose to develop for UPPAAL models.
We also look at sanity checks in a very different area, namely programming integrated development environments (IDEs). As sanity checks have existed for almost 20 years in programming IDEs, and are based on Unix tools where made 40 years ago, we can learn a lot on how to apply sanity checks and make them useful to the user.

4.1. Zeno run detection

Zeno runs occur when infinitely many action occur within a finite time frame. Although Zeno runs are similar to deadlocks, in both cases time cannot pass past a certain point, they are not the same: when in Zeno runs, a transition is always possible, while this is not possible in a deadlock. Therefore, Zeno runs are not detectable by standard deadlock checks that UPPAAL currently supports. In order to detect Zeno runs, liveness checks can be used, which are computationally expensive.

A classical example of a system that contains a Zeno run is the following paradox of Achilles and the tortoise, as recounted by Aristotle:
"In a race, the quickest runner can never overtake the slowest, since the pursuer must first reach the point whence the pursued started, so that the slower must always hold a lead."[10]
Say, Achilles gives a tortoise, which is ten times slower, a 100m head start in a race.

The paradox states that, Achilles must first reach the a point where the tortoise started. During the first step, while Achilles ran 100m, the tortoise crawled a distance of 10m. In the second step, Achilles must run the 10m which the tortoise crawled in the first, during which the tortoise can crawl 1m. One can see that with each step in this race, the distance between Achilles and the tortoise will get divided by 10. As a result, Achilles is only able to approach the tortoise, but not overtake it.
Suppose that the race is modeled in a network of timed automata, then it is trivial that a transition is always possible in which Achilles will reach the point where the tortoise was located. The durations of each step form the geometric sequence $\{a, a0.1^{-1}, a0.1^{-2}, \ldots\}$, where a is the duration of the first step. If we would traverse this system infinitively, the sum of all durations would be finite: $\sum_{n=0}^{\infty} a0.1^n = \frac{a}{1-0.1} = \frac{10}{9}a$.
As we can take infinite transitions within a finite time frame, we are dealing with a Zeno run. Suppose we would write a query in UPPAAL whether or not Achilles is able to overtake to tortoise, UPPAAL would state that this is impossible, even though it is trivial that Achilles could easily overtake the tortoise in the real world.
- Maak ook in related work sectie: schrijf een korte intro met daarin wat voor soort related work je bespreekt.

Looking at this paradox, it becomes clear how Zeno runs can have serious consequences for the analysis of the model that it contains.

Gómez and Bowman[11] have provided an efficient static analysis to prove the absence of Zeno runs in UPPAAL networks. Note that their analysis is not able to prove the presence of Zeno runs, which means that false positives are possible. In order to make the analysis efficient, the original network is transformed to a simplified abstract model, only containing information that is relevant to determining absence of Zeno runs. On the abstract network, liveness checks are able to determine the absence of Zeno runs which are much more efficient than performing liveness checks on the original network.

4.2. Consistency checks in state/event systems

Lind-Nielsen et al.[12] have done research on improving the automatic verification of state/event models. State/event models are concurrent versions of (slightly modified) Mealy machines[13]. State/event models are similar to networks of timed automata, with the exception that state/event models don't model time and don't support

synchronization.
Part of their research was to improve the execution of consistency checks in the commercial tool visualSTATE^{TM}[14]. The consistency checks can be reduced into two categories: reachability and deadlock checks.

The following checks for certain properties (which are relevant to UPPAAL) are done in visualSTATE:

- Unused elements: check whenever a any element in a model is unused (e.g. a state, variable, event)
- State reachability: check whether or not all states in the model are reachable.
- Unread variables/functions/events: check whether or not all variables, functions and events (which correspond to channels in TAs) are read at some point.
- Transition reachability: check whether or not all transition in the model are reachable.ruijters
- Conflicting transition: check whether or not it is possible for a machine to have multiple enabled transitions (which will cause non-determinism in a machine).
- State dead ends: whenever a machine has a state which cannot be left once entered (this check can be disabled).
- Local dead ends: a local dead end is a set of states (a location vector in our terms) that makes a machine unable to change state.
- System dead ends: a system dead end is a set of states that renders the whole system deadlocked (this check can be disabled).
- Arithmetic errors: simple errors like divisions by zero, range errors (over/underflows) or array out-of-bounds errors.

checked These checks are performed based on the guard constraints on the transitions. However, many guards imply other guards. In these cases, many checks can be eliminated: if in some location, there are two guards where $g_1 \rightarrow g_2$, then proving g_1 also proves g_2, which removes the need to check g_2 again.
Through implicational analysis, Lind-Nielses et al., were able to eliminate between 40% and 94% of the reachability checks: if a new guard is to be checked, then first the previously checked guards are looked at that might imply the new guard (a form

of memoization).

We see that many of these checks are also relevant to networks of timed automata, although some properties that are checked (such as dead ends and conflicting transitions) might be modeled on purpose. For example, a system in which 10 parcels are to be processed in a sorting facility *should* be deadlocked after the all parcels have been processed.

4.3. Verification properties in symbolic transition systems

In his work on static analysis of symbolic transition systems (STSs), Sebastaan la Fleur looked at verifying verification properties on STSs[15].
He identified five different properties, of which three can translated to properties in UPPAAL models:

- **Safety properties** describe states that should not be reachable.
  In UPPAAL, this can be simply achieved by queries.
- **Dead transitions** are transitions that can never be taken.
  This corresponds to *unreachable transitions* in UPPAAL.
- **Sinkholes** are reachable states which do not not satisfy the guard of any of the outgoing transitions.
  In UPPAAL terms, this equals a deadlock

4.4. Sanity checks in programming

As sanity checks in UPPAAL are only limited to simple syntax and initialization errors, we look at how sanity checks are provided on other areas.
In programming, sanity checks have been used since 1978, when the Unix tool `lint` was developed that statically checked examined C source code and *"detected features which are likely to be bugs, non-portable, or wasteful"*[3]. The warnings that `lint` provided are now build-in features of most compilers, leaving the original tool to have no use anymore. Nevertheless, most languages spawned their own `lint`-like variants (commonly called linters), which provide sanity checks that are not included in the corresponding compilers. Especially for dynamic and/or interpreted languages like Python or JavaScript, which lack a clear compiling phase in which such warnings can be listed, linters provide a useful tool to catch errors.

With the introduction of modern integrated development environments (IDEs) for programming languages, the functionality of linters where build-in features that require no actions by the user in order for them to work.
As an example, we will look at how the Java IDE IntelliJ IDEA[16] implements sanity checks (called *inspections* in IntelliJ) and which features these sanity checks have.
We identify the following features which make the sanity checks in IntelliJ useful:

- Minimize amount of false positives. When a sanity check returns a warning, it is important that the warning is always valid. When a type of sanity check frequently returns false-positives, one would tend to ignore the warnings.
- Sanity checks are performed on-the-fly. When sanity checks must be triggered by the user (UPPAAL currently uses this approach), the check might get postponed until possibly many errors have been introduced into the model. Performing sanity checks on-the-fly, without requiring to be triggered by the user, errors made by the user will be detected instantly.
- Checks can be disabled on certain parts of the code. In the case of false-positive, e.g. in the case of bad programming practices that are knowingly used, the programmer has the option to silence warnings on certain parts of the code.
- The source of the warning is clearly pointed to. Without directing the user to the source of the error, it is still hard to locate and correct the error.

During development of sanity checks for UPPAAL, the above aspects should be kept in mind on order to make the checks useful for the actual UPPAAL users.

4.5. Conclusion of related work

We have looked at 3 different types related work: work on a single, advanced sanity check in UPPAAL, work on many sanity checks in an other type of timed systems, and the application of sanity checks in other areas.
We can learn from Gómez and Bowman's work that in order to check some properties, the original model must be vastly reduced in order to be efficient. Also, it might be necessary to sacrifice some desired characteristics of a sanity check (e.g. completeness or soundness) for the sake of time/space performance; a solution that is complete, sound and efficient might not exist.
The properties checked in VISUALstate can be directly translated to sanity checks in UPPAAL. Also, Lind-Nielses et al. have presented relevant ways of optimizing the

execution of sanity checks.
Finally, we have looked at an area of application where sanity checks have been extensively used and perfected for 40 years. Many features and aspects that are used in sanity checks in programming IDEs can serve as inspiration for developing sanity checks for UPPAAL.

5. High-level functionality

Before we will present the lower-level architecture and actual implementation, we present the functionality of the tool on a higher level in this chapter. First, we will show the process of finding mistakes in UPPAAL models without a sanity checker. Then we will show the functionality of the sanity checker and how this enhances the debugging of a model.

5.1. Debugging without sanity checker

Without a sanity checker, there are multiple ways a mistake can be detected. We could identify the following scenarios in which a mistake is identified manually:

- **By chance** A mistake could be seen by the user by chance, by looking at the model. Though unreliable, this way the user can directly locate the mistake. However, depending on how many consequences the mistake has on the rest of the model, correcting the mistake may take a long time.

- **By debugging queries** The user may use queries as a way to debug the model. For example, a model might have an 'error'-location that models a situation that should never happen, and thus a query that asserts that such locations are never reached could be used. Note that such methods point the user to a *bug*: the result of a mistake. The next step would be to find the cause of the bug: the mistake itself. Also note that the user is in this case actively trying to find mistakes

- **By observing wrong behavior** The user might expect the model to behave in a specific way. For example: a scheduler should eventually finish its tasks. The user might use the verifier to measure the time it takes to reach this state. Observing that this state is never reached indicates wrong behavior. However, observing wrong behavior might not point clearly as to what caused it.

Note that none of the above methods are both systematic, require low-effort, are effective in pointing to mistakes, or are able to always detect mistakes directly after they are made.

5.2. Debugging with sanity checker

Our tool solves many problems described above by providing sanity checks that are reliable/do not depend on chance and give greater diagnostic information to point to the actual error that was made.
Using the tool comes in two simple steps for the user: choosing/running sanity checks and processing the feedback.

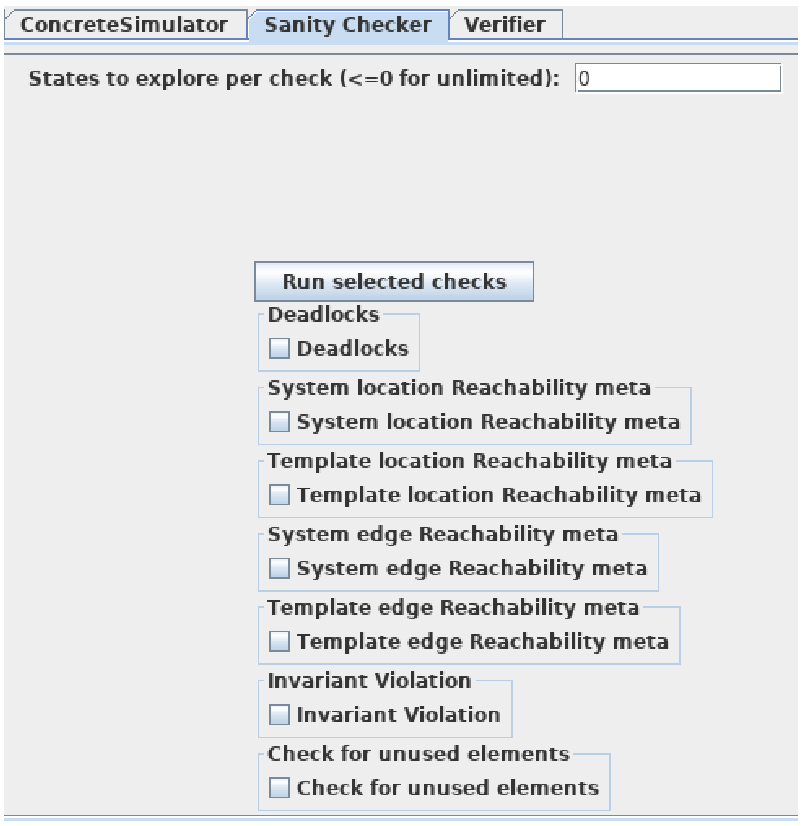
5.2.1. Running sanity checks

The sanity check provides checks for the following properties:

1. **System location reachability** Each location in a system should be reachable.
2. **Template location reachability** Each location in a template should be reachable.
3. **System edge reachability** Each edge in a system should be reachable.
4. **Template edge reachability** Each edge in a template should be reachable.
5. **System deadlocks** As implemented in the prototype, deadlocks occur when no transitions are enabled in a state.
6. **Component deadlocks** Similar to system deadlocks, component deadlocks occur when no transitions are ever enabled within a component. *This sanity check could not be implemented.*
7. **Invariant violation** After an enabled transition is executed, it should never happen that an invariant of a location in the target state is violated.
8. **Unused language declarations** Whenever a variable/channel/function is declared but never used in the model.

GUI

The user can select the wanted checks in a separate tab built into the GUI of UPPAAL, as shown in Figure 10. Using the button, the sanity check will run all selected sanity checks.



States to explore per check (<=0 for unlimited): 0

Deadlocks
- [ ] Deadlocks

System location Reachability meta
- [ ] System location Reachability meta

Template location Reachability meta
- [ ] Template location Reachability meta

System edge Reachability meta
- [ ] System edge Reachability meta

Template edge Reachability meta
- [ ] Template edge Reachability meta

Invariant Violation
- [ ] Invariant Violation

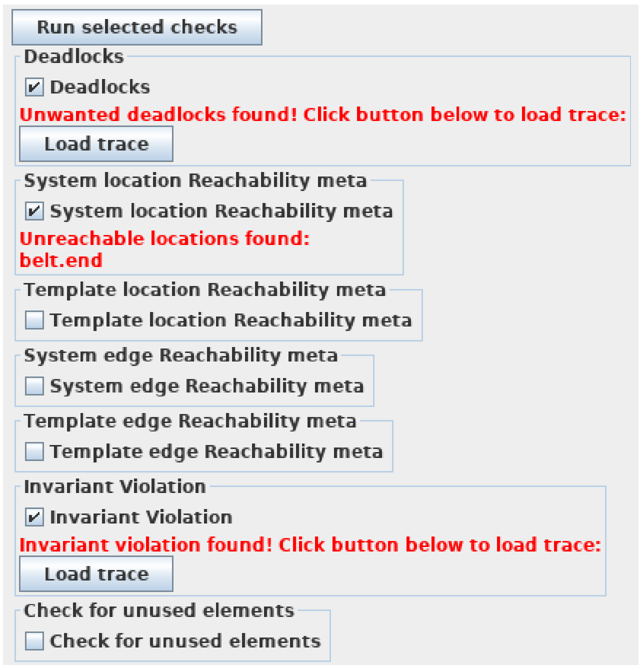
Check for unused elements
- [ ] Check for unused elements

Figure 10.: Selecting sanity checks in the sanity checker tab.

5.2.2. Feedback

Feedback is provided in three ways, textually in the sanity checker tab, graphically in the template editore, or through diagnostic traces.

Feedback in the tab is very basic, but gives a quick indication if anything un-



Run selected checks

Deadlocks

✓ Deadlocks

Unwanted deadlocks found! Click button below to load trace:

Load trace

System location Reachability meta

✓ System location Reachability meta

Unreachable locations found:
belt.end

Template location Reachability meta

☐ Template location Reachability meta

System edge Reachability meta

☐ System edge Reachability meta

Template edge Reachability meta

☐ Template edge Reachability meta

Invariant Violation

✓ Invariant Violation

Invariant violation found! Click button below to load trace:

Load trace

Check for unused elements

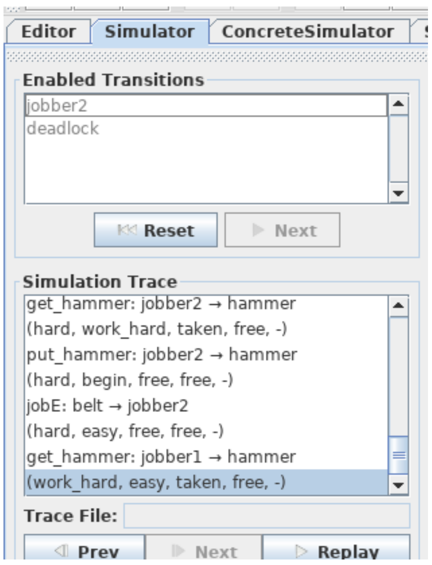
☐ Check for unused elements

Figure 11.: Feedback in the tab.

usual has been found. In the case of deadlocks and invariant violations, a button is shown that loads the trace in the simulator.

A trace bears very helpful diagnostic information for the user: it shows the exact transitions that point to a certain violation (e.g. deadlock or invariant violation). In Figure 12, we see a loaded trace to both a invariant violation (the greyed-out enabled transition of *jobber2* indicates an invariant violation) and a deadlock.

Finally, in Figure 13, we see graphical feedback in the template editor. For system location/edge reachability, red indicates that the location/edge is unreachable in all processes implementing that template, while yellow means that the location/edge is reachable in at least one, but not all processes of that template.



```
jobber2
deadlock
```

```
get_hammer: jobber2 → hammer
(hard, work_hard, taken, free, -)
put_hammer: jobber2 → hammer
(hard, begin, free, free, -)
jobE: belt → jobber2
(hard, easy, free, free, -)
get_hammer: jobber1 → hammer
(work_hard, easy, taken, free, -)
```

Figure 12.: A trace to a deadlock and a invariant violation.

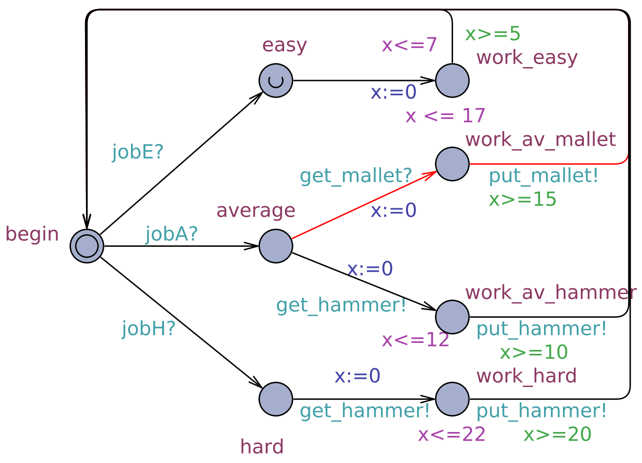


Figure 13.: Graphical feedback in the editor

5.3. Conclusion

Looking at the functionality of the sanity checker shows us many improvements when comparing this to the process of debugging models without the sanity checker.
This functionality gives the user a effective tool that solves the problems when debugging without a sanity checker:

- The sanity checker systematically covers the whole model, and not just some specific parts of it. Where a human user might only look at some specific places for mistakes, the sanity checker will look at the complete state space of the model for errors.
- The sanity checks are low-level, meaning that they look at individual locations and edges. One might say knowing that a single edge or location is unreachable bears more diagnostic value than observing a specific bug (e.g. observing that a scheduler never terminates all tasks). Also, invariant violations and deadlocks are low-level bugs which could make them easier to understand, especially when provided with a trace.
- Using the sanity checker requires little user effort: selecting the wanted checks and running the sanity checker with a hotkey is enough. Therefore, the user may be inclined to use the sanity checker more often than he/she would otherwise actively search for mistakes.

6. Architecture

This chapter presents the the overall architecture of the sanity checker, specifically how the tool combines existing components with new additions presented in this thesis.

6.1. Overall architecture

From a black-box perspective, the input of the tool is the UPPAAL model, which is in the form of a `Document` internally (see Section 3.3.2), and the selected properties that are to be checked. The output is visual feedback in the GUI of Uppaal.

Internally, the tool consists of existing, and new components. The overall architecture

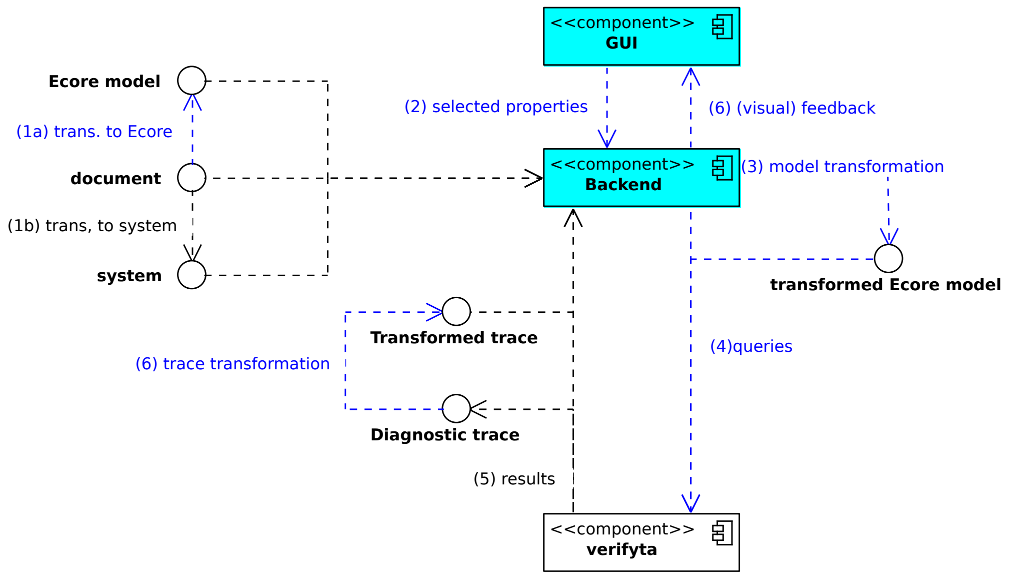


Figure 14.: The overall architecture of the tool. Blue elements are new components, other elements are existing components.

can be seen in Figure 14.

The main procedure of the sanity checker can be divided in the following steps, as visualized in Figure 14:

1. After the user has started the sanity checker, the document will first be transformed to two other representations: to the Ecore model (which also compiles the declaration language), and it will be compiled to a `UppaalSystem`..
2. The user has a selection of properties that has to be checked. Each property may or may not have options in them that can be changed. The properties can be selected by the user in a separate tab in the GUI of UPPAAL.
3. For each property, the tool may apply model transformations to the Ecore model, resulting in a *monitoring* UPPAAL model for that specific property. Note that before a model will be given to the verifier, the resulting model will be translated to a `UppaalSystem`, see Figure 15
4. For each property, possibly multiple queries will be generated which UPPAAL will check. We will use the model API to execute queries.
5. The verifier will return results for each property, which either proves or disproves the query along with a diagnostic trace that serves as a counter example where possible.
6. If a diagnostic trace has been provided by the verifier, then this trace belongs to the *monitoring* UPPAAL model. In order to make the trace conforming to the original model, the trace will be transformed.
7. Finally, the results are presented to the user textually in the tab of the sanity checker, and graphically in the editor of Uppaal.

6.2. Components

We will now look into each aspect of the sanity checker individually. Some components already exist, as indicated in Figure 14.

Uppaal model representations and translations

Strictly speaking, each Uppaal model has four different representations: plain-text XML, a `Document` or a `UppaalSystem` inside the UPPAAL model API, and the Ecore model.
Each of these representation have different purposes (and limitations) within the

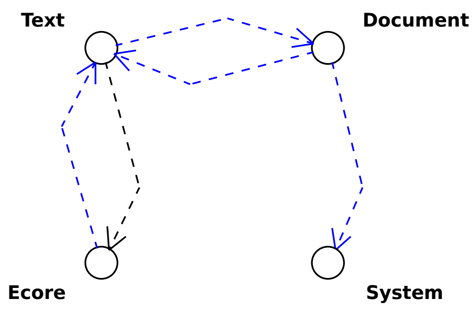


Figure 15.: Possible representations of a Uppaal model, and the existing translations between them (in blue) and the new translation (in black).

sanity checker and UPPAAL:

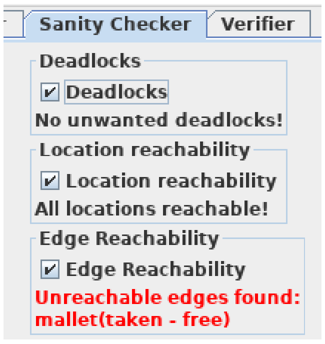
- **Plain-text XML** allows Uppaal to save a model to the file system. This even includes models that have syntax errors or cause other compiler errors. The obvious limitation is that it's not practical to work with plain-text models programmatically.
- The **Document** model is a structured representation based on the XML structure. This model is used in the Uppaal editor: locations, edges, templates are modeled in this representation. However, the declaration language remains plain-text. This allows for syntax/compiler errors to be present in the declarations, as long as the XML structure conforms to the XML schema for Uppaal models.
- An `UppaalSystem` is a model that has been compiled (implying that no syntax/compiler errors are present). This representations most importantly contains the actual processes (a process represents a TA) of the model and a compiled list of variables and clocks. Note that this representation is linked to the `Document` model; it could therefore be seen as an argumentation of the **Document** model. This representation is used by Uppaal in the simulators.
- In the Ecore model, every aspect of an Uppaal model is represented, including the declaration language. This will enable us to fully analyze the model. This representation also allows us to transform models easily.

We use existing and new translations to translate one representation to an other. The following translations are used so get to/from the representations (see Figure 15):

- **Document to System** This translation is included in the model API included in UPPAAL. This translation is irreversible as information about parametric templates and edge selects is lost. The elements (templates, locations etc.) of the resulting `UppaalSystem` contain references to their origin in the `Document`.
- **Document to Text and Text to Document** These translations are also included in the model API.
- **Text to Ecore** This translation did not exist previously. This transformation is done using a parser generator called *Xtext*[17]. This way, the plain-text XML file can be directly compiled to an Ecore model.
- **Ecore-Text** This translation exists as part of the Ecore meta-model of UPPAAL.

GUI

The front-end of the sanity checker consists of a new tab added to the GUI of



No unwanted deadlocks!
All locations reachable!
Unreachable edges found:
mallet(taken - free)

Figure 16.: Simple textual feedback in the sanity checker tab.

UPPAAL. In this tab, the user can select the properties that should be executed when the sanity checker is requested to run. Results of checks will be presented textually in this tab (see Figure 16) and, if possible, graphically in the editor by changing the `Document` (e.g. through coloring locations/edges, see Figure 17).

Ecore model transformations For some sanity checks (e.g. edge reachability), simply using queries isn't sufficient. In these cases, the original model needs to be transformed in order for the sanity check to be possible. For example, Figure 18 shows the jobber model from Figure 17 after transforming it in order to make invariant

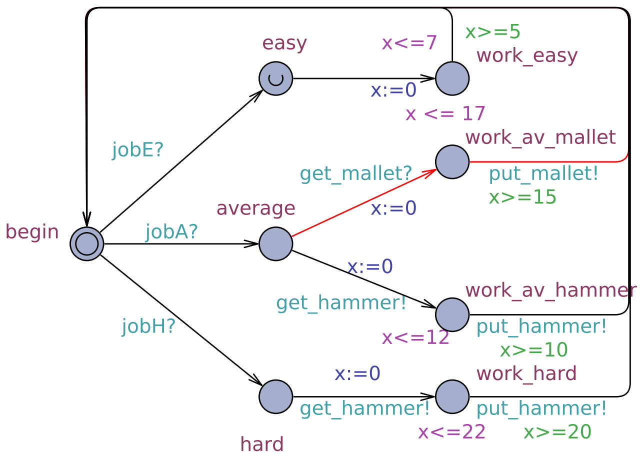


Figure 17.: Graphical feedback in the editor

violations detectable. These transformations can be applied on the declaration language level (e.g. to make a line coverage checker), or on the template level (e.g. adding extra locations). The transformation can be done programatically on the Ecore model.

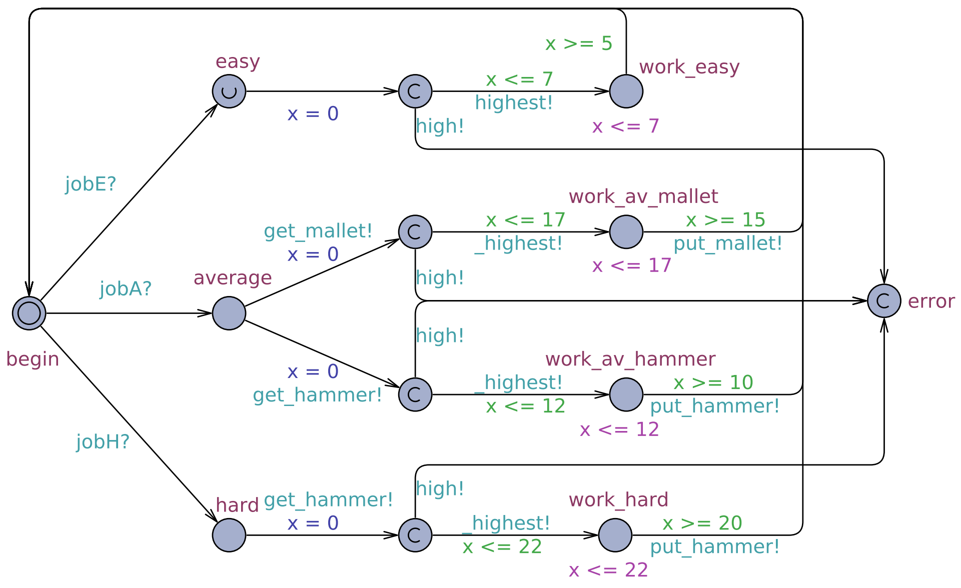


Figure 18.: Transformed jobber model for invariant violation detection

Verification Using the model API, the (possibly transformed) model will be given to the verifier which will evaluate the queries. The Ecore model will undergo three

translations before it can be given to the verifier: firstly the Ecore model will be serialized to plain-text XML, then the plain-text will be read by the model API which returns a `Document`, and finally this will be compiled to a `UppaalSystem` which can be given to the verifier.
After verification, there may be a diagnostic trace (e.g. a trace that leads to a deadlock). This trace is a Java Object provided by the model API and can be transformed if necessary (in order to leave out redundant states/variables/locations/templates from the transformed model).

Implementing a sanity check In order to implement a new sanity check, a subclass of the abstract class `AbstractProperty` must be implemented. Inside this class, an UPPAAL model will be supplied on which the implementation can apply model transformation and/or static analysis.
Specifically, this abstract class requires a single method `check()` to be implemented that gets all representations of the original model. After the sanity check has been done, this method can call a callback, providing it with an `SanityCheckResult` which can either be outputted textually to an output stream or graphically to the GUI as a `JPanel`. In addition, feedback can be given through coloring elements in the `Document` that has been provided.

7. Sanity checks

In this chapter we will show the implementation of the tool, as well as the approach for each sanity check.
The tool is implemented in Java, using packages from the Eclipse Modeling Framework (EMF). Most importantly, Ecore is used as a base as the metamodel is written in it[9].

7.1. Selected sanity checks

In this section, we will present the implementations for each sanity check that are included in the tool. As an UPPAAL model example, we will refer mostly to the factory model as described in 3.2.
The following properties have been selected for which we have implemented sanity checks (unless stated otherwise):

1. **System location reachability** Each location in a system should be reachable.
2. **Template location reachability** Each location in a template should be reachable.
3. **System edge reachability** Each edge in a system should be reachable.
4. **Template edge reachability** Each edge in a template should be reachable.
5. **System deadlocks** As implemented in the prototype, deadlocks occur when no transitions are enabled in a state.
6. **Component deadlocks** Similar to system deadlocks, component deadlocks occur when no transitions are ever enabled within a component. *This sanity check could not be implemented.*
7. **Invariant violation** After an enabled transition is executed, it should never happen that an invariant of a location in the target state is violated.

8. **Unused language declarations** Whenever a variable/channel/function is declared but never used in the model.

Below, we describe each property in detail.
In particular, we provide a formal definition for each property, describe possible implementations and their transformations and argue about the soundness and correctness of the implementations.

7.2. Implementations

7.2.1. System location reachability

Definition

The first type of location reachability is system location reachability. System reachability dictates that every location of every process in a Uppaal system should be reachable.

Definition 7.1 (System location reachability). *Consider a network $A = \langle A_1, \cdots, A_n \rangle$. System location reachability requires that for every component A_i, and every location in that component $l \in L_i$, were $1 \leq i \leq n$, there exists an reachable state $\langle \bar{l}, v \rangle$ for which $l \in \bar{l}$ in A.*

For example, in Figure 19 we have a parameter p and template-local variable a. Say that two processes P0 and P1 are defined with $p = 0$ and $p = 1$ resp. It is trivial that L0 is reachable for both processes as it is the initial location. L1 is only reachable for the second process due to the guard on p. L2 is not reachable for both processes due to it's guard. And trivially, L3 is not reachable due to the absence of incoming edges. Thus, the unreachable locations are P0.L1, P0.L2, P0.L3, P1.L2 and P1.L3.

Naive implementation

A very simple way to check location reachability for a single location is using a simple query:

```
E<> (P0.L1)
```

This query checks whether a state is reachable in which the `belt` is in the `end` location. However, it is not possible to check multiple location in a single query. Therefore, the implementation of the sanity check will generate a reachability query

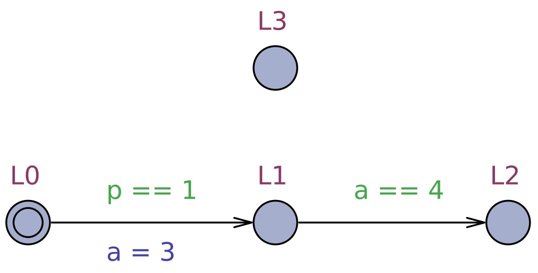


Figure 19.: Example of unreachable locations

for each system location, and will give them to the verifier. Hence, this method is expensive for the verifier [1]
Finally the results are presented to the user. Unfortunately, to disprove reachability, UPPAAL has to exhaust the state space. As a result, no helpful diagnostic trace can be presented to the user if location reachability is not proven.
Note that this approach does not use model transformations.

Meta variable implementation

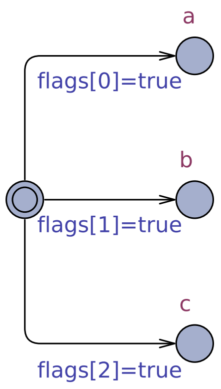


Figure 20.: Setting meta variables on entering a location

A more sophisticated approach to doing location reachability checks is by using the concept of meta variables.
The problem with the first approach is that a query is needed for each location. For simple models in which all locations are reachable, this is not a problem. However,

[1]The verifier is able to reuse the state space between queries, thus saving time by only having to generate the state space once. However, each query still has to be evaluated for each state, which is still expensive.

when a location is not reachable, UPPAAL can only prove this by evaluating the query on the entire reachable state space, which is computationally expensive for larger models. For models with both a large state space and many locations, exhausting the entire state space for every unreachable location is very inefficient.
In this implementation, we add a boolean *flag* variable, that has been declared meta, for each location: when the location is entered, the *flag* variable gets turned on (e.g. boolean set to true) using the update of the incoming edge. The value of this flag is preserved during the whole verification (see 3.3.1 for the behavior of meta variables). Say we have the template in Figure 20 for which we use a meta array of three boolean flags `flags`. We can now solve location reachability in a single query:

```
E<> (flags[0] && flags[1] && flags[2])
```

It is trivial that there does not exist a single path in which all locations have been visited. Thus, if `flags` were a normal non-meta array, the above query could not have been satisfied as it is not possible for all flags to be true.
However, declaring the array as a meta variable will cause the following steps during verification (contents of array are included for each step):

1. Initial state. `flags=[false, false, false]`
2. Visit location `a` (deadlock). `flags=[true, false, false]`
3. Backtrack to initial state and visit `b` (deadlock). `flags=[true, true, false]`
4. Backtrack to initial state and visit `c` (deadlock). `flags=[true, true, true]`
5. Query satisfied, end verification.

If one of the locations is unreachable, the entire state space had only been visited once, making this approach far more efficient.

However, as meta variables are not part of the state of the system, they are not part of the trace that is given as output. This means that if the verifier evaluates the query to false, we have no way of telling which locations were not visited. This makes this solution not *effective* in checking location reachability.

One way to view the values of meta variables is by copying them to normal variables as part of the system state. However, this would cause state space explosion which would make this solution less efficient. For example, including flags for 20 locations in the system state could increase the state space up to 2^{20} times.

To solve this problem, we use one of the properties of meta variables: between verifications, the values of meta variables are kept. This means that after the first verification (which queries the flags of the locations), the values of these flags are kept on subsequent verifications. We can use a second verification that only copies the meta variables to normal variables so that they appear in the trace.
To copy the values of meta variables to normal variables, we add an additional initial location to each template. This initial location has an outgoing edge to the original initial location. On traversal of this edge, the meta variables will be copied to local variables, which are available in the trace. This edge will also synchronize with an broadcast channel sent by an added process. This ensures that the system will start with a single transition which copies all meta variables to local variables. This does not enlarge the state space, as the local variables are not updated afterwards. To read the variables, we use a query that will evaluate to true once the first transition has been fired. The trace of this query will contain the flags indicating which locations are reachable.
Note that changing options between verifications cause meta variables to be reset. As such, the main verification must have the trace option enabled, which introduces an extra delay as a (potentially) long trace must be read.
See Figure 21 for the transformed model. The array *_fm* is a template-local meta int array of flags that indicate which location where reached. On the first transition, the values in this array are copied to a normal array *_f*, the values of which appears in the trace.

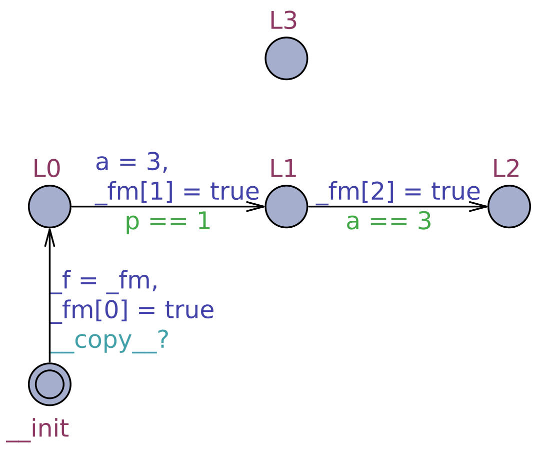


Figure 21.: Copying meta variables to the state on first transition

Meta variable implementation 2

A possible problem with the above meta variable implementation is that the added non-meta, although never being changed, increases the size of the state. For larger models with many system locations, this can cause an otherwise small state to become very large.
An other solution to reading the values of meta variables is by querying them. However, to prevent the verifier to exhaust the entire state space again, we introduce a Stopper template. This stopper template uses a single edge from a committed state to a location with the invariant *false*. This forces UPPAAL to try this edge first, but disables it to take it as the target invariant is. The edge is guarded by a `stop` variable. The model API enables us to set a initial state before using the verifier. For the normal verification as described in the meta variable implementation, we set the stop variable to `false`; this causes the original behavior of the model. After verification if not all locations are reachable, we use the verifier for every system location. However, we now set the stop variable to `true`, causing that no transitions can be taken. This ensures that running the verifier for every location is done very fast.

Output

The output of the implementations is a list of system locations that are not reachable. The results are presented to the user textually in the sanity checker tab. We also present the result graphically by coloring the non-reachable location red or yellow in the template editor, depending one whether all instances of a location are unreachable, or some. See Figure 22.

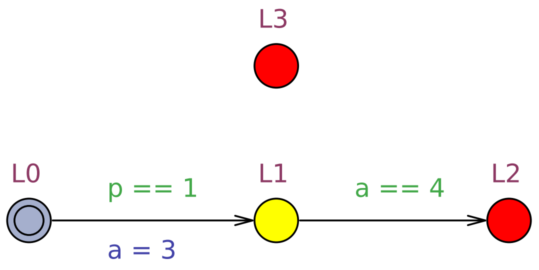


Figure 22.: Coloring unreachable locations

Soundness & completeness

In models whose state-space is easily exhausted, the sanity check for location reachability is sound. Reasoning behind this is simple: an location not visited during exhaustive verification is by definition not reachable.
For models whose state-space can only be partly visited, the sanity check is not entirely sound. As not all states of the model can be visited, some locations might also not become visited that might be in fact reachable.

Regardless of the size of the state-space, the sanity check for location reachability is complete: if a location is not reachable, it will never be reached during verification, regardless of whether or not we visit the whole state-space.

7.2.2. Template location reachability

Definition

Template location reachability dictates that every location of a template should be reachable by at least one process based on that template.

Definition 7.2 (Template location reachability). *Let a network $A = \langle A_1, \cdots, A_n \rangle$ have a partition $\mathcal{S}$ of it's component indices $\{1, \cdots, n\}$. The elements of $\mathcal{S}$ are index sets of components that belong to the same template.*
Assume that the set of locations of a component be ordered: if components A_1 and A_2 belong to the same template, then $l_{1,i}$ and $l_{2,i}$ are instances of the same location in the template.
Template location reachability requires that for every index set J in $\mathcal{S}$ belonging to a template, and every location index i for that template, there exists a component A_j, with $j \in J$ for which there exists a reachable state $\langle \bar{l}, v \rangle$ in A for which $l_{j,i} \in \bar{l}$.

If we look at the template in Figure 19, it is trivial that L0 is reachable, L1 is reachable in P1, L2 is never reachable and L3 is also never reachable. Concerning template location reachability, only L2 and L3 are unreachable.

Naive implementation

The naive implementation is almost equal to the naive implementation of system location reachability. However, for templates that have multiple components based on it, we are satisfied when only one component reaches a location of their template. For example to check for reachability of *hard* in Figure 7 we can check for either jobber1 or jobber2 to be in that location:

```
E<> (jobber1.hard or jobber2.hard)
```

This is repeated for every location in all templates.

Meta variable implementation

This implementation is identical to the meta variable implementation of system location reachability.
However, the array that contains the flags indicating reachability are not local to components, but are declared globally. Entering the same location, but in different processes, will flip the same flag.

Output

The output is similar to template reachability, the difference being that locations are color black (default color) when at least one process can reach them. The textual output in the sanity check tab identifies the edge using the name of the template, instead of the name of the process (e.g. Jobber.hard instead of jobber1.hard).

Soundness & completeness

Soundness of template location reachability is the same as system location reachability: the sanity check is sound if the entire state space could be explored.
Also, the sanity check is complete for the same reason as system location reachability.

7.2.3. System edge reachability

Definition

System edge definition dictates that every edge of every process of a system must be reachable. Before we make this formal, we define what it means when an edge is taken:

Definition 7.3 (System edge reachability). *Consider a network $A\langle A_1, \cdots, A_n\rangle$. System edge reachability requires that for every component A_i, and every edge in that component $e \in E_i$, there exists an transition t reachable in A that selects e.*

For example, in Figure 19, the edge to $L1$ is only reachable in P1, and the edge to $L2$ is never reachable.

Implementation

We check for edge reachability the same way as we check for location reachability using meta variables. The difference being that every edge sets a flag belonging to *itself*, rather than its *target location*. Note that checking for edge reachability is not possible without using model transformations.
Even more, after checking for edge reachability, we can deduct the reachable locations based on which edges were reachable; a location is reachable if and only if it has an incoming edge that is reachable (w.e.o. initial locations). Therefore, it is useless to check for location and edge reachability separately; if both checks are enabled, only edge reachability is checked, after which location reachability will be deducted.
Even though this might make checking for location reachability only sound obsolete, checking for location reachability remains less time/memory consuming. This is trivial as checking for edge reachability terminates when all edges are reachable, while all locations might have been visited before that happens.

Output

The output of the sanity check is a list of system edges that are not reachable. The result of the sanity check is, like location reachability, presented textually in the sanity checker tab, and visually in the editor. To be consistent with location reachability, we color edges that are reachable in no process red, and those that are in some but not all processes blue. See Figure 23.

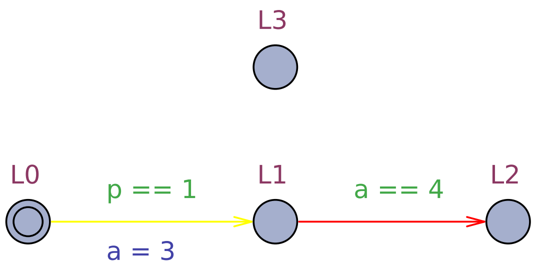


Figure 23.: Coloring unreachable edges

Soundness & completeness

Comparable to location reachability, models whose state-space is easily exhausted, the sanity check for edge reachability is sound.
For models whose state-space can only be partly visited, the sanity check is not entirely sound, because of the same reason checking for location reachability is not sound.

Similarly to location reachability, the sanity check for edge reachability is complete: if an edge is not reachable, it will never be reached during verification, regardless of whether or not we visit the whole state-space.

7.2.4. Template edge reachability

Definition

Template edge reachability dictates that for every template, every edge in that template should be reachable in at least one process.

Definition 7.4 (Template edge reachability). *We use the idea of the partition $\mathcal{S}$ from 7.2.2 to define this property formally.*
Let the set of edges of a component be ordered: if components A_1 and A_2 belong to the same template, then $e_{1,i}$ and $e_{2,i}$ are instances of the same location in their template. We let $e_{i,j}$ be the j'th edge of the i'th component of the network. Consider a network $A = \langle A_1, \cdots, A_n \rangle$ and the partition $\mathcal{S}$ indicating processes built from the same template. Template edge reachability requires that for every index set J in $\mathcal{S}$ belonging to a template, and every edge index i for that template, there exists a component A_j, with $j \in J$ for which there exists a reachable transition $\rightarrow$ for which $\rightarrow$ selects the edge $e_{j,i}$.

For example, in Figure 19, the edge to $L1$ is reachable through P1, while the edge to L2 is never reachable.

Implementation

The implementation for this sanity check is trivial: at the core, we have the implementation of system edge reachability using meta variables. To convert this to template edge reachability we apply the same changes needed for template location reachability.

Output

Similarly to template location reachability, the difference with system edge reachability is that edges that are only colored black if they are not reachable in any process.

Soundness & completeness

Comparable to all other reachability properties: when the whole state space is explored, the implementation is sound, otherwise not.
Comparable to all other reachability properties, the implementations are complete.

7.2.5. System deadlocks

Definition

System deadlocks occur when a system is in a state out of which there are no *enabled* transitions, except for delays . This can be considered a liveness property for the whole system: eventually, something can eventually happen in the system.

Definition 7.5 (System deadlock). *A deadlock is defined as a reachable state s in a network A for which no outgoing reachable enabled action transition $s \overset{\epsilon}{\Rightarrow} s'$ exists.*

For example, a deadlock occurs in Figure 19 when being in L1; no transition is possible as the only outgoing edge is disabled.
However, in some cases a deadlock is wanted. For example when a system was meant to have 'end' build into to it. We therefore define *wanted* deadlocks as deadlock states that have at least one location in its location vector that has no outgoing edges. Trivially *unwanted* deadlocks are deadlocks in a state for which every location in its location vector has an outgoing edge.
Formally, a deadlock in a state $s = \langle \bar{l}, v \rangle$ is *unwanted* when for all $l \in \bar{l}$ there exists an outgoing edge $s \rightarrow$.
This sanity check aims to detect *unwanted* deadlocks.
For example, the deadlock described in 19 is unwanted, as it does not occur in a location without outgoing edges. If we look at the factory model in 3.2, we could say that a wanted deadlock will occur when the conveyor in Figure 6 is in its final state.

Implementation

As the query language of UPPAAL already has the `deadlock` keyword that indicates whether or not a system is in a deadlock, implementing this sanity check for all deadlocks is simple:

```
A[] (!deadlock)
```

To accommodate this query to ignore wanted deadlocks, we first analyse the model by collecting all locations without outgoing edges (e.g. `belt.end` for the factory model). In words, we require that a deadlock is only accepted in a state that is in one of these locations. This translates to the following query for the factory:

```
A[] (deadlock imply belt.end)
```

One example for which this is not possible is when an 'end'-location does not have a name. This is simply solved by giving this location a temporary name, allowing us to refer to it in the query, removing this name afterwards. We apply this directly to the Document loaded in UPPAAL, as this reduces the overhead involved in recompiling an Ecore model back to a Document and to a UppaalSystem.

Note that UPPAAL does not handle deadlock queries in models with guarded broadcast receivers due to state space explosion. Reasoning behind this is that 10 guarded broadcast receivers may spawn 2^{10} possible transitions, as UPPAAL must choose every possible combination of enabled receivers.

Output

When a deadlock has not been found, the output of the query is simply positive. It is reported in the tab to the user.

However, when a deadlock has been found, the verifier provides a trace to this deadlock. Although one could say that we did transform the input model by adding location names, the trace is linked to the original Document and UppaalSystem because of the way the Model API works. Consequence being that when we reset the names in the Document, the trace will 'know' about this.
The Model API provides a way to convert a trace to a textual format which we can present in the tab. In the future, it might also be possible to load the trace into the simulator itself [2].

[2]Through reverse engineering, it is already possible to programmatically load the trace into the simulator

Soundness & completeness

Given that the Uppaal verifier is correct, every detected deadlock is an actual deadlock. Furthermore, this can always be proven by the diagnostic trace that it provides.

When the model's state space can be exhausted, we can ensure that a deadlock will be found if one exists. In that definition of completeness, we can say that the implementation is complete. However, one might argue that the implementation can only detect one deadlock at a time, letting subsequent deadlocks stay undetected until the first has been resolved.
When the state space can not be exhausted, the implementation is not entirely complete. Furthermore, in models with guarded broadcast receivers, the Uppaal verifier does not allow deadlock predicates in queries. This disables us to detect any deadlocks.

7.2.6. Component deadlocks

Definition

A component deadlock happens in a component when no transition is *reachable* that involves that particular component. This can be considered a liveness property for a single component: something will eventually happing in a component.
Component deadlocks are more sophisticated than system deadlocks as these cannot be detected by built-in properties like the `deadlock` keyword. In plain English, we must prove that a component *could* always eventually traverse an edge.
However, this becomes more difficult when we mix in dead ends (locations without outgoing edges), because the above property will falsely consider a state in `belt.end` as a component deadlock. We could include dead ends into the informal definition of component deadlocks as follows: *The situation in which a component can not eventually traverse an edge, unless the component is in a dead end.* This will take away the false component deadlock for the `belt` component, but the sanity check will still trigger for other components (e.g. the jobber). For example, a state will be eventually be reached where the belt is depleted and no outgoing transitions are enabled, without the jobbers and the tools being in a dead end. Thus this will falsely be seen as a component deadlock for the tools and jobbers.
Thus, we change the definition to the following: *The situation in which a component will not eventually traverse an edge, unless the system is in a state with a dead end.*.

However, this still leaves us with an unwanted edge case: if the belt only has an easy job left on it, the edges in the tools become unreachable as easy jobs don't require tools. Thus, the tools are in a component deadlock without there being a dead end in the state.

We've now come to a point where 'filtering out' the edge cases on dead ends in component deadlocks make the definition less and less helpful. One could reason about the usefulness of detecting *unwanted* component deadlocks in such systems: if such systems have an intentional end build into them, then surely it might be wanted that some components are at some point *done*, even if no other component is in a dead end. The boundary between *wanted* and *unwanted* component deadlocks is hard to express in a sanity check and might be prone to false positives/negatives which make the sanity check even more unhelpful. Therefore we will disable component deadlock checks for *finite* systems (systems with a dead end in a component).

Now that we are only considering infinite systems without dead ends, we can consider the approach to doing this sanity check.

The sanity check translates to the following statement: every component can always eventually involve in a transition. We can formalize this as follows:

Definition 7.6 (Component deadlock). *Let $A = \langle A_1, \cdots, A_n \rangle$ be a network. A component deadlock happens in a state s reachable in A, when which there exists an component index i, where $1 \leq i \leq n$, for which there exists no transition t reachable from s that selects an edge in E_i.*

In terms of queries, this comes down to the following query for a single component:

```
A[](true imply E<>(/* component involved in transition */))
```

It is, however, not allowed to nest quantifiers inside each other. The above query, if completed, would thus result in a syntax error.

Uppaal does provide us with 'leads to' operator that is almost equal to the above query. The query `a --> b` is equal to the following query:

```
A[](a imply A<>(b))
```

However, the semantics of this query are slightly different to the query we would like to use: the wanted property states that a something *may* eventually happen, while the above query states that something *will* eventually happen. The difference is obvious in the factory example, as it is perfectly fine for the `mallet` to never be used (which would violate to the above query), even though it is *possible* for it to be used (which satisfies the wanted property).

There are two ways in which this problem can be solved. Firstly, we could transform

the model in a way such that it *must* eventually involved in a transition, be it within a very large clock bound. However, this would change the semantics of the original model and might yield false positives, which would make the sanity check unhelpful. Secondly, we could approach the sanity check by querying if the component can always participate in a large amount of transitions: surely if it at least can be in do 1000 transitions then we could conclude that it wouldn't be deadlocked afterwards. This, however, might result in false negatives: a component that *does* deadlock after 1000 transitions would stay undetected.
Given the restrictions in implementing this sanity check, we can conclude that no *helpful* sanity check can be implemented for component deadlocks using the current features of the query language.

7.2.7. Invariant violations

Location invariants are meant to restrict the time the system can stay in particular locations, which formally means that they are meant to restrict *delay* transitions. However, invariants might also prevent a system to fire an *action* transition due to the target state's invariants. In this situation, this transition's successor state is considered *unreachable*.
However, restricting action transitions is not the purpose of invariants; edge guards serve this purpose. It is therefore considered bad practice to have invariants that can be violated after firing enabled transitions.

Definition

Informally this sanity check must proof that there does not exist an enabled transition (see 3.1.3), whose target state violates an invariant.

Definition 7.7 (Invariant violation). *For a network A, an invariant violation occurs when an enabled transition $s \overset{\epsilon}{\Rightarrow} \langle \bar{l}, v\rangle$, reachable in A, exists, where $v \not\models I[\bar{l}]$.*

Implementation

First, we need to translate this to a situation which Uppaal can solve. As Uppaal ignores enabled transitions whose successor states violate an invariant, we need to transform the original model in such a way so that such transitions *can* be taken, *and* so that they can be detected through verification. Also, we must not enable/disable new behavior after transforming the model that would change the semantics of the original model (and thus lead to false positives/negatives).

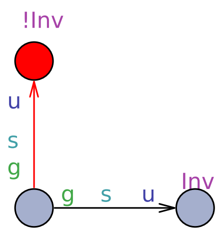


Figure 24.: Concept of detecting an invariant violation
Red elements are added in the transformation

We present the approach of detecting invariant violations in Figure 24. We transform the model by making a copy of every edge, replacing it's target location with a new location that has the same invariant of the original target, but negated.
Whenever we are in the starting state, the model might continue with it's normal behavior by going right. It can, however, also traverse to the edge going up. This can only happen when the original transition was enabled, but in addition the invariant would be violated in it's successor state. Simply checking if the added location is reachable would complete the sanity check for invariant violations.

However, invariants are restricted as per Definition 3.5. One restriction is that invariants must not contain lower bounds in clock constraints. Thus, negating an invariant such as $x \leq 5$, where x is a clock, would after reduction result in $x > 5$, which is not a legal invariant. Therefore, the approach as shown in Figure 24 would result in an invalid system.
We solve this problem through a new transformation shown in Figure 25. We move

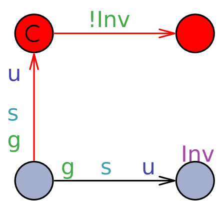


Figure 25.: Improved concept of detecting an invariant violation
Red elements are added in the transformation

the negated invariant to the guard of an extra added edge. This extra edge points to an new 'error'-location (which is universal to the template). To prevent a delay in the top-left location, we make it committed. This way, the behavior of the the transformed model remains the same.

However, there are two problems with this approach. Firstly, there are situations in which this approach would cause false positives. Consider the case in which a process is in the upper-left location in Figure 25 by synchronizing with an other process that would also be in a committed location (not a location added through transformation). This other process might fire an action that would enable the upper edge in Figure 25 which might not be enabled before. This would be detected as an invariant violation, even though the invariant was not violated *immediately* after taking the first transition.
Secondly, guards are also restricted in a way that only conjunctions of certain expressions are allowed (see Definition 3.4). An invariant such as $x \leq 5$ negates to $!(x \leq 5)$, which does not qualify as a valid guard directly without simplifying. Furthermore, more complex invariants as $x \leq 5 \wedge y < 5$ which negate to $!(x \leq 5 \wedge y < 5)$ can not be reduced to valid guards.

Thus, we need to make sure that we do not allow other actions while in the upper left location in Figure 25. In order to do this, we can use channel priorities, a feature in Uppaal intended to remove non-determinism that allows us to prioritize certain channels above others. This in turn enables us to ensure that the outgoing edges of the upper-left location are always taken first before others.
Using this approach we are able to detect invariant violations in a simple way. See

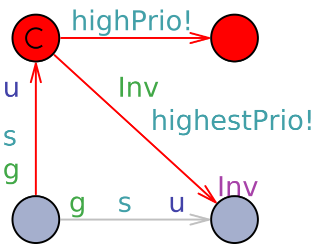


Figure 26.: Using channel priorities to detect invariant violations
Red elements are added in the transformation. The original edge (grey) is removed.

Figure 26 for the new transformation.
When in the upper-left committed location, the original transition is enabled. The channels *highestPrio* and *highPrio* are given the highest and second highest priority respectively. If the invariant evaluates to true **immediately**, then the diagonal edge

must be taken due to the priority on it's synchronization. This is equal to the original behavior. Whenever the invariant is violated, the diagonal edge is not enabled, and thus we must take the upper edge as it has the second highest priority. This indicates an invariant violation, which can be detected by the verifier.
Note that we can safely remove the original edge, as the added locations and edges can also model the original behavior of the edge. Keeping the original edge would increase the state space of the verification. Also note that the upper-right 'error'-location is only needed once per template. This means that every invariant would introduce one new location and two new edges after the transformation, in addition to a single 'error'-location for each template.

Transformations

This sanity check uses two transformations: model and trace transformations.

Firstly, the model transformation transforms the source model as visualized in 26. Concretely, it redirects the every edge to a location with an invariant to a new, committed location. From this new location, an edge to an erroneous location is added, which synchronizes on the second-highest priority channel. An edge is also added to the original edge's target, synchronizing on the highest priority channel. The algorithm is shown in Appendix A.
When a invariant violation is detected, the verifier returns a trace leading to the 'error'-location. However, this trace belongs to the transformed model. In order for the diagnostic trace to be helpful, we must convert it back to make it conform to the original model.
To do this, we make use of how the trace is constructed internally. We make use of some implementation details:

- In any translation shown in Figure 8, the order of edges, locations, templates is preserved; if no element (location, edge etc.) is removed, the position of an element in it's parent remains the same.
- The model transformation only adds locations/edges and redirects a single edge (as shown in Figure 26).
- Added locations/edges are appended to the end of the location/edge list.
- The trace object that is provided by the Model API links to system edges and

location which all have an index that indicate their position in their template (i.e. index i means $(i + 1)$'th edge/location).

Knowing these details, we can construct an efficient way of transforming the trace to conform to the original model. The main idea is simple. First, replace the location vector in every state in the trace with a location vector using locations of the original model (using their indices). Similarly, replace the edges contained in a transition with the edges belonging to the original model. If the location vector contains an added location, discard it and skip to the next state.
In pseudo-code, it comes down to the following (for details on the transition and system structure, see [6]):

```
Input: ts : list of transitions, sys : original system
Output: transTs : transformed list of transitions
SymbolicState prev = null;
SymbolicState curr = null;
SymbolicTransition[] result = []; Iterator it = ts.iterator();
outer:
while (curr = it.next()) != null do
    SystemEdgeSelect[] selectedEdges = curr.getEdges();
    // first transition is empty
    if prev != null then
        // insert original model edges into selectedEdges
    end
    inner:
    while true do
        if curr.getTarget() contains added locations then
            if it.hasNext() then
                curr = it.next();
            else
                break outer;
            end
        else
            SymbolicState origTarget;
            // copy curr to origTarget with original model locations
            result.add(new SymbolicTransition(prev, selectedEdges, origTarget);
            prev = origTarget;
        end
    end
end
```

Algorithm 1: Transforming transition

Output

As with deadlocks, invariant violations can be proven not to exists, in which case we will report this in the tab, or are proven to exists by a trace. In the last case, the trace is transformed to conform to the original model, after which it can be presented the same way as with deadlock traces.

Soundness & completeness

To prove soundness, we first prove that actions transitions to locations with invariants keep original behavior when no invariant violations are present.
To do this, we first deduce the order of operations in a action transition from the definition of the transition relation in 3.1.3. The order of executing a transition is as follows:

1. Check that the variable valuations conforms to all the edge guards selected in the transition and check if the transitions conforms to the priorities as specified by rules of committed locations and channel/process priorities. Passing these checks indicates that the transition is *enabled*

2. Apply the updates of the edges to the variables. In the case of synchronizations, the sending edges update fires first. In the case of broadcast synchronizations, the receivers' updates are fired in order of declaration in the system declaration statement.

3. Assert that the updated variable valuations conform to the target locations' invariants. Failing this assertions indicates an invariant violation.

It is trivial that the first step is unaltered by the transformation; nothing about the edge (except its target) and source location has been changed.
The second step is also preserved, as the update of the edges are not changed. Also, the order of updates remains the same.
The way that the third step is transformed is more complicated. It is sufficient to prove that when the state is reached where we are in the original target location, no other transitions have been fired, other than transitions from the new committed location to the target location. Assuming that the updated valuation conforms to the targets invariant, the only enabled transition are from the new committed location to the target location as this edge has the channel with the highest priority. Therefore, as long as not every component has left the new committed location, no other transition can happen. Therefore, the third step of executing a transition is preserved.
Thus, the transformation does not change behavior of the original model if no invariants are violated. Therefore, the implementation will not detect invariant violations if there are not any, which proves soundness.
Also note that soundness is preserved if the entire state space could not be explored.
To prove completeness, we must prove that invariant violations are always detected if they exist.

We already know that the original behavior is preserved if no violations are encountered. We must prove that once an invariant violation can occur, it will be detected.
It suffices to prove that we can detect that in the third step of executing an transition, it will be detected that one of the target state's invariants is violated.
Thus, we assume that we are in the new committed locations while the target location's invariant Inv evaluates to true. In this trace, only the edge to the new erroneous location is enabled. Because this edge's channel currently has the highest priority, this edge will be taken. Detecting a state to be in the erroneous location is sufficient to detect this violation. Therefore, whenever a invariant violation is reachable, a component will reach the erroneous location which will be detected. This proves completeness.

7.2.8. Unused language declarations

Definition

The declaration language used in Uppaal features basic syntax checking. However, it does not check for declarations that are not used. This property dictates that any declaration of variables, clocks, channels or functions must be used somewhere in the model.
Specifically, using a declaration means that an identifier linking to that declaration is included somewhere in the Uppaal model.

Implementation

To implement this, we can use the functionality of the Eclipse Modeling Framework (EMF), in which the meta-model of Uppaal is made. One functionality of the framework is that the elements of a model are linked with references. This means that, given a variable, we can find all usages of that variable in the model. The same can be used for any declaration in the language: variables, functions, channels or clocks.
EMF also makes it possible to easily construct a qualified name for a variable so that the user can easily identify the variable. The following algorithm shows the approach of this sanity check, in which we can clearly see the power of the framework:

```
**Input:** nsta : ecore input model
**Output:** unused : set of qualified names of unused variables
let unused = [];
foreach EObject obj : nsta.allContents() do
    if obj instanceof Variable then
        if EcoreUtil.UsageCrossReferencer.find(obj, nsta).isEmpty() then
            let name = obj.getName();
            while obj.getContainer() != null do
                obj = obj.getContainer();
                name = obj.getName() + "." + name;
            end
            unused.add(name);
        end
    end
end
```

Algorithm 2: Collecting unused declarations

Note that we have not yet incorporated the verifier in this sanity check. This would be considered a more *static* type of sanity check. Also note that we check if the declarations are used in the model; we do not check if they are actually read/written to.

Output

We present the list of qualified names of unused declarations only textually in the sanity checker tab, as we do not yet have access to highlighting errors in the textual editor in the GUI.

7.3. Limitations

When using the UPPAAL verifier to execute our sanity checks, the whole state space might be exhausted to give sound and complete results. For less complex models, the state-space may be visited within seconds. However, for more complex models, visiting the whole state space is impossible; in general, the the state-space increases exponentially in relation to the size of the model.

One consequence is that we can not wait for the verifier to terminate, which might sometimes happen only if the whole state-space has been visited. Terminating the verifier by disconnecting would mean that meta-variables are reset. This means

that location and edge reachability require a built-in mechanism that terminates verification after a certain amount of time. Therefore, a counter meta-variable is added to the transformed model that keeps track of the amount of transitions that the verifier has taken. On every 'action'-edge (an edge representing an internal action or an edge sending on a channel), we increment the counter. By adding a guard to every edge on the counter (e.g $counter < 10000$), we limit the amount of transitions the verifier can take during verification.
However, not visiting the whole state space means that locations or edges might not be visited, even though they are reachable. Also, possible invariant violations or deadlocks might remain undetected.

Also, depending on whether a sanity check must exhaust the state space, the search order matters for performance: breadth-first is preferable if the entire state space must be exhausted, depth-first is preferable otherwise (e.g. when the query is (dis)provable by a trace). However, the sanity checker cannot know upfront which will be the case. We therefore assume correct models and use the appropriate search order: depth first for edge/location reachability (we want to reach that), breath first for deadlock/invariant violations (we don't want to reach that).

7.4. Conclusion

In total, 13 implementations are made for 7 sanity checks: both location reachability checks have 3 implementations, both edge reachability checks have 2 implementations and the other sanity checks have one.
Table 1 presents a summary of all implementations, showing for each implementations which techniques are used (model/trace transformations, Ecore, queries).

We can see how there are many different types of implementations in the sense that they use different techniques in order to do the sanity check. Most interesting is the implementation for invariant violations, which uses all techniques.

		used techniques				
sanity check	impl.	Ecore	queries	model-trans	trace	trace-trans
system locations	naive		✓			
	meta-1	✓	✓	✓		
	meta-2	✓	✓	✓		
template locations	naive		✓			
	meta-1	✓	✓	✓		
	meta-2	✓	✓	✓		
system edges	meta-1	✓	✓	✓		
	meta-2	✓	✓	✓		
template edges	meta-2	✓	✓	✓		
	meta-2	✓	✓	✓		
system deadlock			✓		✓	
invariant violation		✓	✓	✓	✓	✓
unused declarations		✓				

Table 1.: Summary of implementations and the techniques it uses

8. Evaluation

In this chapter, we will evaluate whether or not the implemented sanity checks conform to the wanted qualities: soundness, completeness, efficiency and effectiveness. If a sanity check has multiple possible implementations, we will end with a conclusion for which implementation is preferable.
Note that each implementation has two scenarios: one in which a state is reachable that will end verification (e.g. a deadlock), and one in which such state does not exist (e.g. model without deadlocks) and thus requires exploring the entire state space. In the first scenario, the time it takes to reach such a state depends on the model, it's size and even the order of edges and locations in it's XML representation as this influences the order in which the state space will be explored. As such, one cannot compare verification times of two different models, even if they have the same size and structure. Even more, simply changing the order of locations in the source file, which semantically yields the same model, can change verification time. This effect can especially be seen when using depth-first search.
Therefore, for efficiency testing, we will use the train-gate model that is included in Uppaal [4]. This model can be easily scaled up by adding more trains, without changing the underlying model. This allows us to compare performances of verification runs with different trains.
All tests were done on an Intel® Core™ i7-5600U CPU @ 2.60GHz (max turbo freq. 3.2GHz) system with 16GB RAM (1600MHz) running 64-bit Linux (Debian 9.6).
In Appendix B, the test results belonging to each graph are shown in tables.

8.1. State space exhaustion baseline

In order to evaluate the performance of implementations when exhausting the entire state space, we first construct a baseline: what is the time/memory performance of the original/untransformed model. This can be compared to the performance of the implementations when exhausting the state space. We will use both breadth-first search and depth-first search as they are both used depending on the sanity check.

See Figure 27 for the results. Note that both search approaches use the same amount of memory.

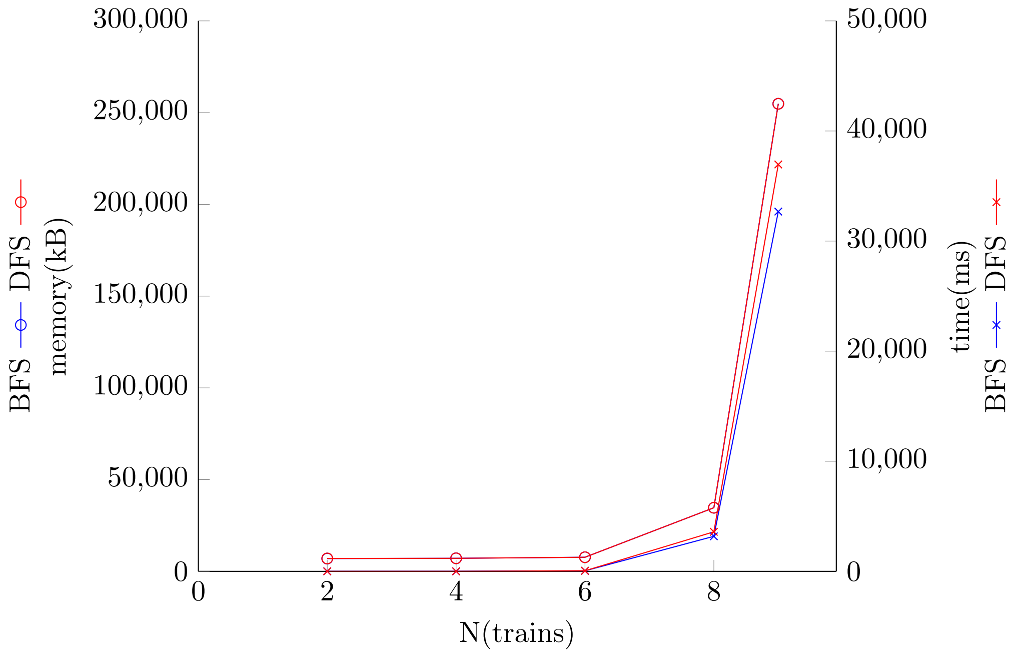


Figure 27.: Time/space performance state exhaustion using breadth- and depth-first search (BFS/DFS resp.)

8.2. System location reachability

8.2.1. Efficiency

We will now evaluate the memory and time efficiency of location reachability.
The three possible implementations, with and without meta-variables, will be compared. We will measure used memory and the elapsed time.

In Figure 28 we can see the performance of both implementations for increasing amounts of trains in the model, when all locations are reachable.
The naive approach performs predictable: it's performance worsens fast as it the model size increases as it has to do a verification for each location. We especially see the drastic memory increase to more than 3GB. At 100 trains, the verifier runs out of memory.
We see that the first meta variable implementation performs drastically better in

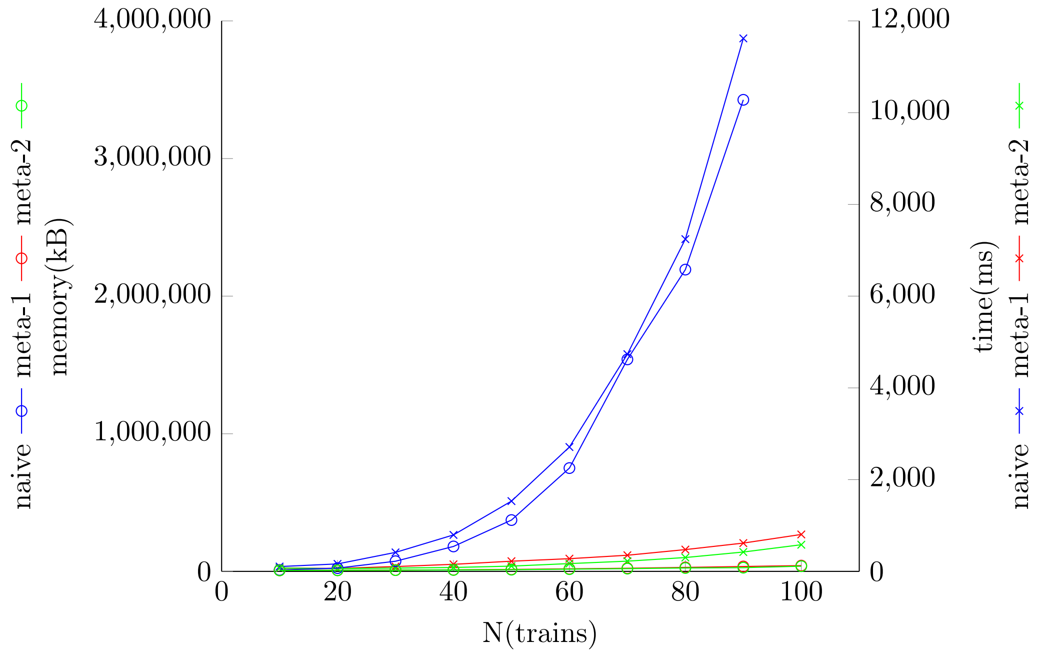


Figure 28.: Time/space performance of system location reachability (all locations reachable)

both time and memory.
The second meta variable implementation sees even more time/memory performance increase. Thus, we can conclude that it is more efficient to query each location flag separately, than to copy the meta variable arrays into the state.

Next we will introduce an unreachable location, effectively forcing the verifier to exhaust the state space.
See Figure 29 for the results. As expected, performance decreases drastically when not all locations are reachable as the whole state space must be explored.
Looking only at memory, both meta implementations clearly perform better.
However, when looking at time performance the second meta variable clearly performs worse than the others. Further inspection revealed that, contrary to expectations, the final queries on the meta variables take a long time when the corresponding location was not reached. Although we force the final queries to be started on a state without outgoing transitions, it still seems that the verifier needs to explore the entire state space again.

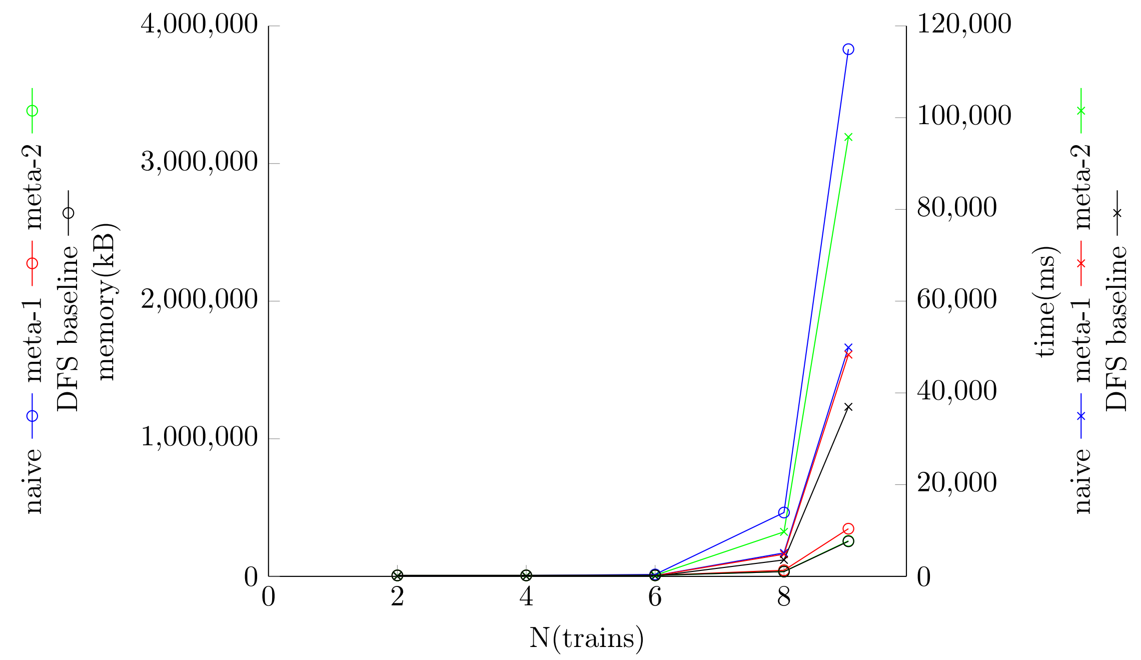


Figure 29.: Time/space performance of system location reachability (one locations unreachable)

8.2.2. Effectiveness

All implementations are very effective in the sense that it returns the exact information the user needs to pinpoint the unreachable locations.
The only way to make it more effective is by providing more information as to *why* the location was unreachable. For example, by giving a trace that would end on a location close to an unreachable location. For now, we consider this step outside of the scope of this thesis.

8.2.3. Conclusion

For now, we can conclude that the first meta variable implementation gives the best results. Even though that, given all locations reachable, it performs slightly worse than the second meta variable implementation, it is twice as fast a job in discovering unreachable locations (when there are any). The naive implementation is simply too slow and memory expensive compared to the other implementations.

8.3. Template location reachability

8.3.1. Efficiency

We will now evaluate the memory and time efficiency of location reachability.
The three possible implementations, with and without meta-variables, will be compared. We will measure used memory and the elapsed time.

In Figure 30 we can see the performance of all implementations for increasing

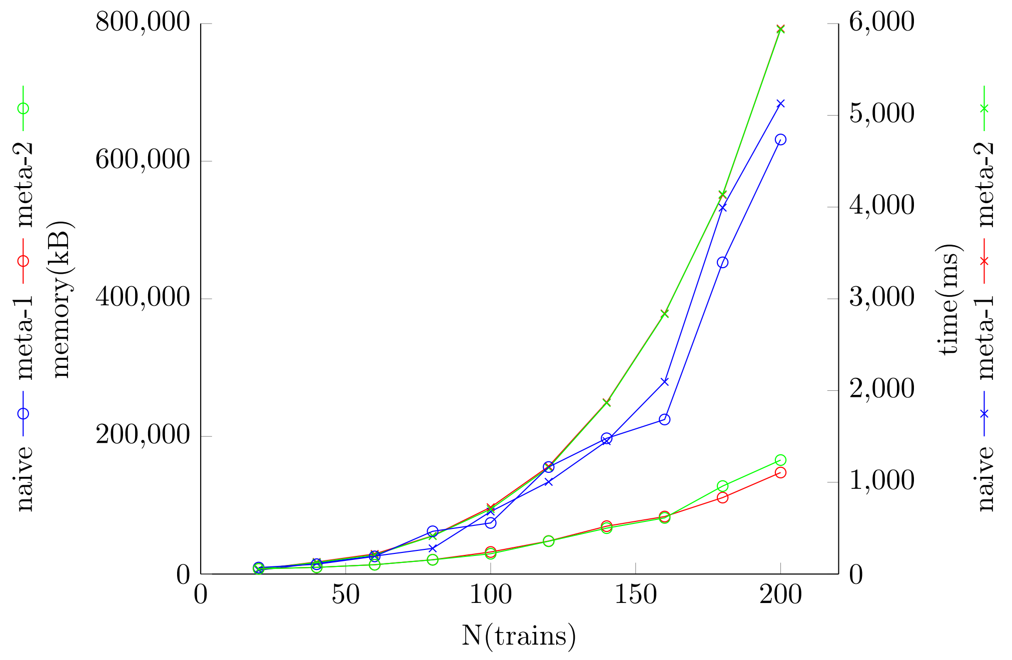


Figure 30.: Time/space performance of template location reachability (all locations reachable)

amounts of trains in the model, when all locations are reachable.
The time performance of all implementations are very similar, although the naive implementation shows slightly better results. This can be explained by the fact that the amount of queries remains the same as the number of trains increases.
For memory performance, we see that both meta implementations show much better memory usage. There is no simple explanation for this, other than that UPPAAL reuses the state space between queries which might fill up memory.
We can also see that the performance of both meta implementations are very similar, compared to the differences seen for system location reachability in Section 8.2. This

can be explained by the fact that the second meta implementation gained efficiency by removing the large boolean array from the state, while the array stays small for template location reachability. Also, the explored state space is very small as the amount of locations that have to be visited remains the same.

Next we will introduce an unreachable location, effectively forcing the verifier to exhaust the state space.
See Figure 31 for the results. We can see very similar results as with system location

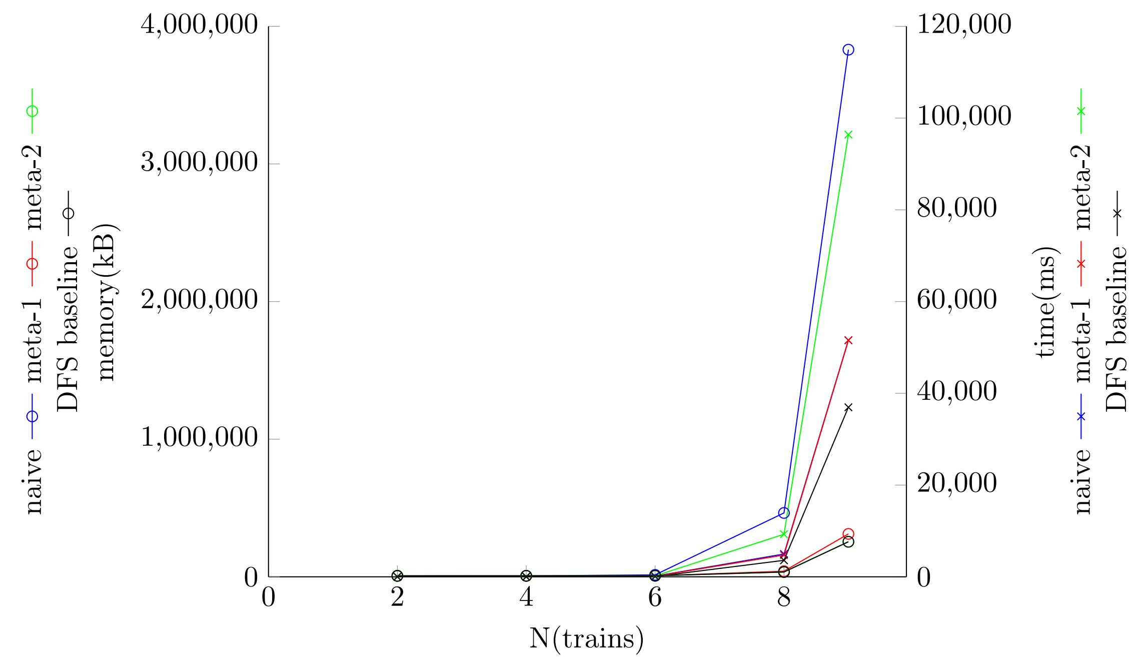


Figure 31.: Time/space performance of template location reachability (one location unreachable)

reachabilit 8.2.1 which is expected: both checks result in exploring the entire state space if a location is not reachable.
What's remarkable is that we would expect a significant time decrease for the first meta variable implementation compared to system location reachability as the size of the state has drastically decreased. We can see the memory usage has decreased. However, time performance has worsened (although this might be by chance). This might indicate that the larger state size does not worsen the time performance significantly.

8.3.2. Effectiveness

The implementations are similarly effective compared to the system location reachability implementations.

8.3.3. Conclusion

We can first conclude that we can turn down the second meta variable implementation as it has significant time performance decrease when there are unreachable locations.
Next, when comparing the other implementations, we look at the bad memory usage of the naive implementation: nearly 4 GB for 9 trains.
Based on that, we can conclude that the first meta implementation is preferable.

8.4. System edge Reachability

8.4.1. Efficiency

We will now evaluate the memory and time efficiency of edge reachability.
The two possible implementations, with and without meta-variables, will be compared.
We will measure used memory and the elapsed time.

In Figure 32 we can see the results of the efficiency tests.
We can clearly recognize the results of location reachability in Table 29 and 31, which where results of models with an unreachable locations. Thus, we could conclude that in order to reach all edges in the model, nearly the whole state space had to be explored. It is of course expectable that edge reachability takes longer to prove than location reachability; visiting all location does not mean all edges have been visited.
Next we will introduce an unreachable edge, effectively forcing the verifier to exhaust the state space.
See Table 33 for the results. We can see very similar results with previous models in which the entire state space was explored, although with a slight a slight time performance decrease. This can be due to the fact that the query contains more flags to be checked.

8.4.2. Effectiveness

Similar to location reachability implementations, the implementations for template edge reachability are very effective: they give exactly the information needed to

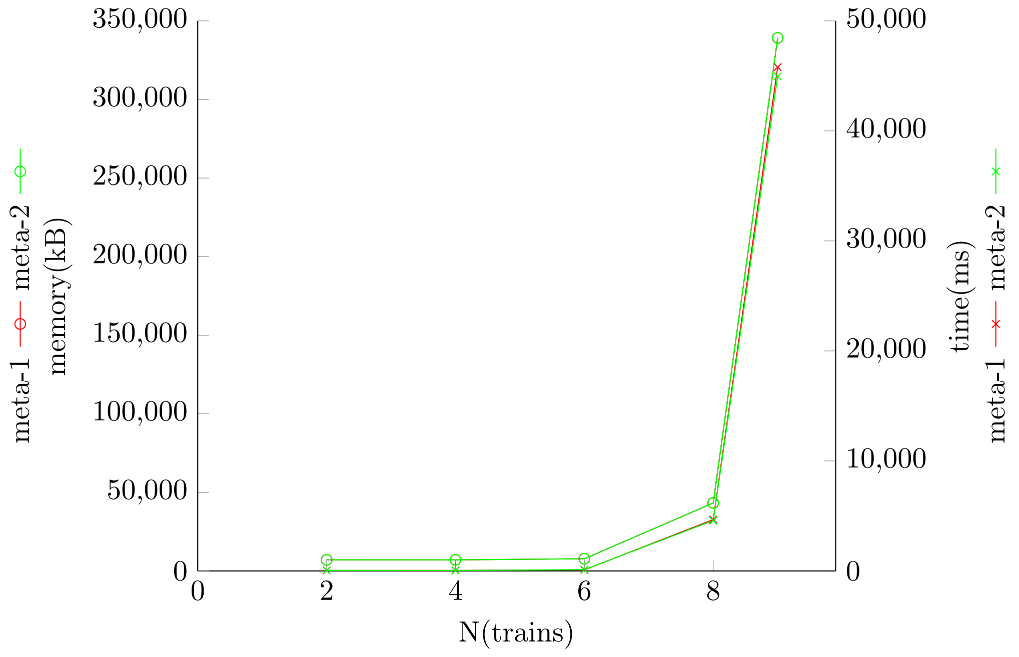


Figure 32.: Time/space performance of system edge reachability (all edges reachable)

diagnose the problem.

8.4.3. Conclusion

Due to the fact that the second meta implementation takes twice as long when there exists an unreachable edge, the first meta implementation is preferable.

8.5. Template edge Reachability

8.5.1. Efficiency

We will now evaluate the memory and time efficiency of template edge reachability. The two possible implementations, with and without meta-variables, will be compared. We will measure used memory and the elapsed time.

In Table 10 we can see the results of the efficiency tests.
Like template location reachability, we can see that the implementations are very fast compared to system edge reachability. Both implementations show similar time/memory usage, with the second implementation performing slightly better.

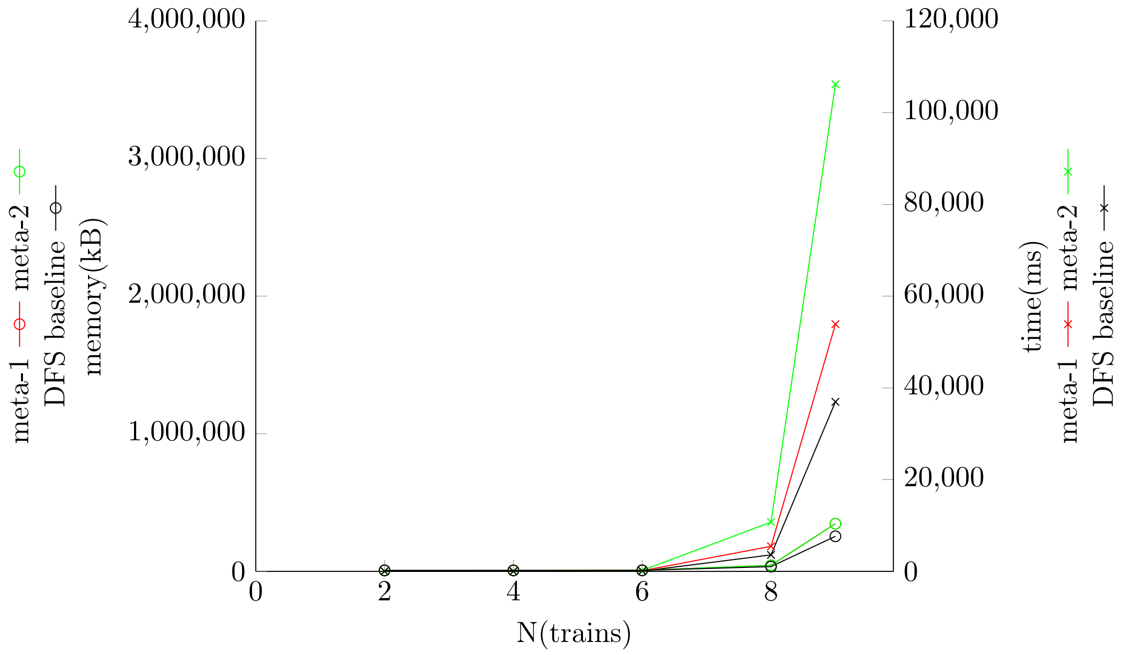


Figure 33.: Time/space performance of system edge reachability (one edge unreachable)

Next we will introduce an unreachable edge, effectively forcing the verifier to exhaust the state space.

See Figure 35 for the results. As expected, the results are very similar to previous meta implementations that exhaust the state space.

8.5.2. Effectiveness

Similar to location reachability implementations, the implementations for template edge reachability are very effective: they give exactly the information needed to diagnose the problem.

8.5.3. Conclusion

When all edges are reachable, the second meta implementation performs slightly better. However, based on the far worse time performance of the second meta implementation when not all edges are reachable, we can conclude that the first meta implementation is preferable.

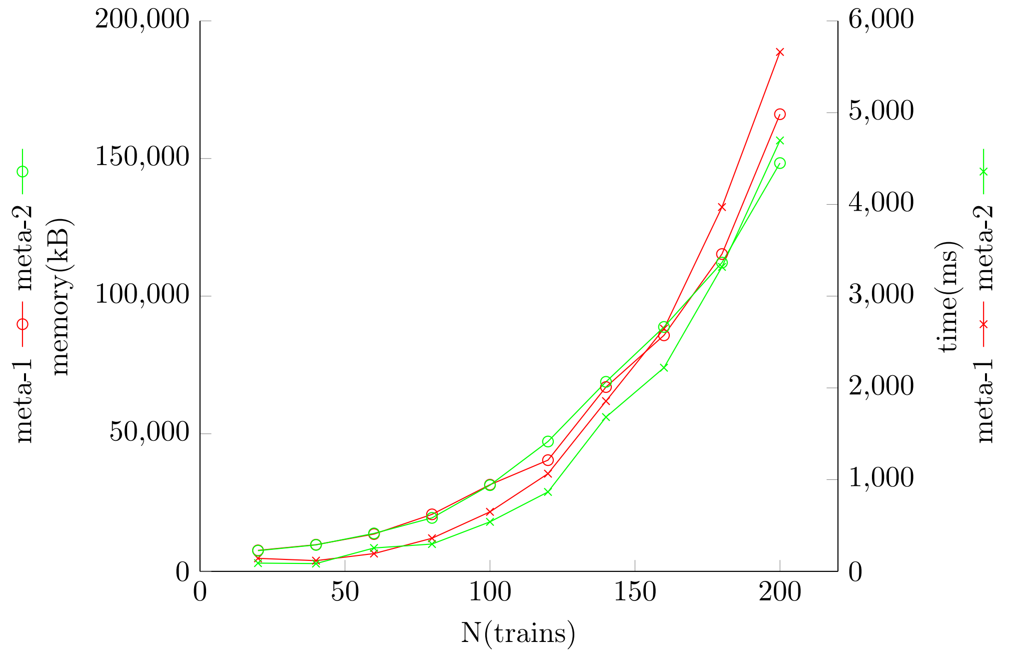


Figure 34.: Time/space performance of template edge reachability (all edges reachable)

8.6. System deadlocks

8.6.1. Efficiency

Due to the train-gate model having guarded broadcast receivers, we are unable to test the detection of deadlocks in this model. However, as the implementation does not require model transformations, and the query for deadlocks is simple, we can assume time/memory performance of comparable to exhausting the original model as done in 8.1.

8.6.2. Effectiveness

A deadlock is state without outgoing action transitions. The best diagnostic for such state is a trace that leads to that state, which is exactly what the sanity check provides. The trace contains all information necessary to understand how the deadlock was reached. Thus we can conclude that the implementation is very effective.

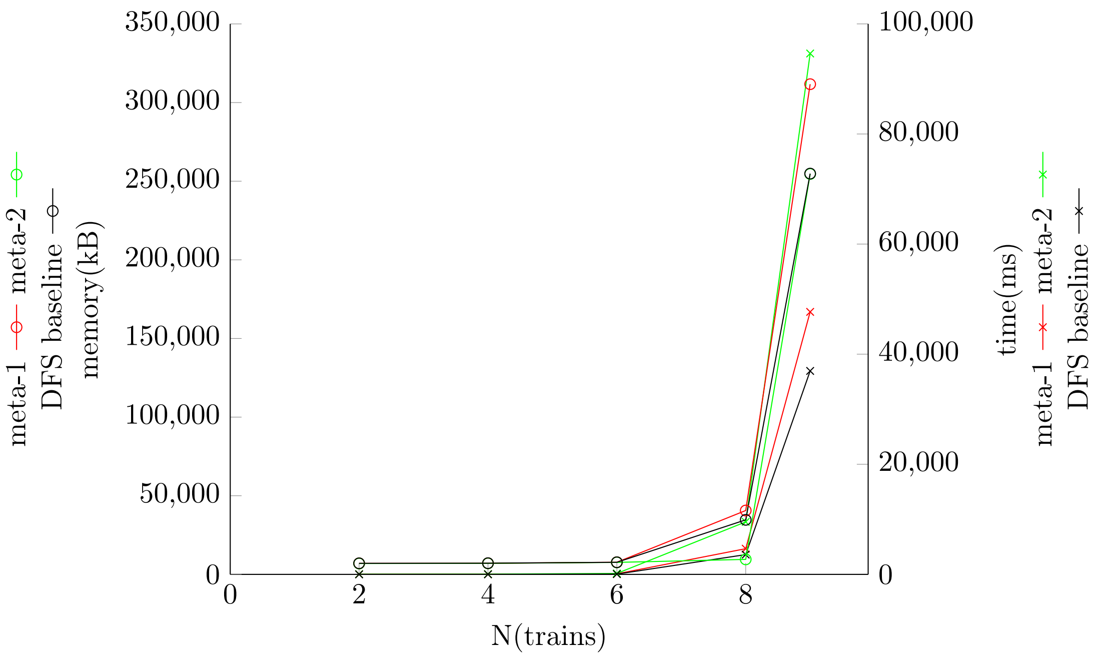


Figure 35.: Time/space performance of template edge reachability (one edge unreachable)

8.7. Invariant violations

8.7.1. Efficiency

First, we test the efficiency for the unmodified model which has no invariant violations. This causes the verifier to exhaust the state space of the modified model.

We can see the results in Figure 36.
When comparing the performance to the baseline performance as shown in 8.1, we can see that both time and memory usage approximately doubles. This was to be expected: most locations in the model have invariants, and transitions to locations with invariants will be represented by two transitions in the transformed model.

We now introduce a human error that can lead to an invariant violation. Concretely, we remove the clock reset on the edge between `Train.Start` and `Train.Cross` (see [4] for more details on the model).

See Figure 37 (blue plot) for the results. Time usage stays within a minute when

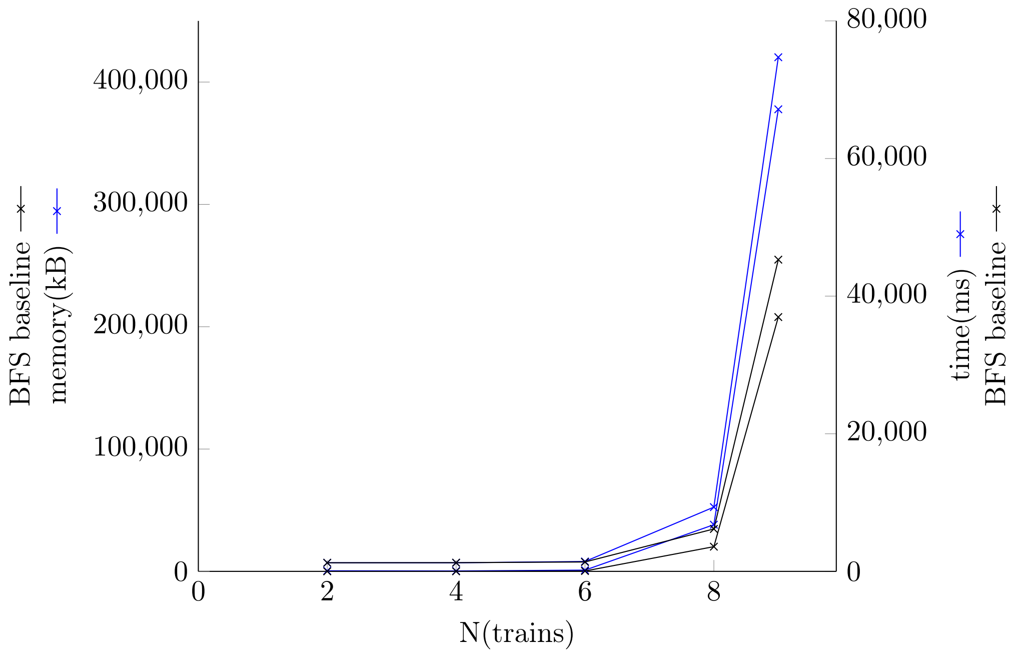


Figure 36.: Time/space performance of invariant violation reachability (no violations reachable)

30 trains are used.

However, we can compare this test with the template edge reachability test reachability: it can be proven that the invariant violation happens when all edges have been fired. Reasoning behind this is that the edge between `Train.Start` and `Train.Cross` can only be fired when another train has crossed the gate without stopping, which implies a state before which all template edges have been fired at least once. Still when we compare the performance of the invariant violation implementation to the template edge reachability implementations in Figure 34, we see that the invariant violation implementation performs far worse.

This is caused by the fact that invariant violation uses breadth-first search (which is faster in exhausting the state space (which happens when the model is correct). Edge/location reachability uses depth-first-search, which is faster in reaching witness-states for a query (states that (dis)prove a query).

Using depth-first search instead yields the results as shown by the red plot in Figure 37. We see very similar results to template edge reachability as shown in Figure 34. However, when using depth-first search, the provided trace is very long (±400 transitions for the 200-train model).

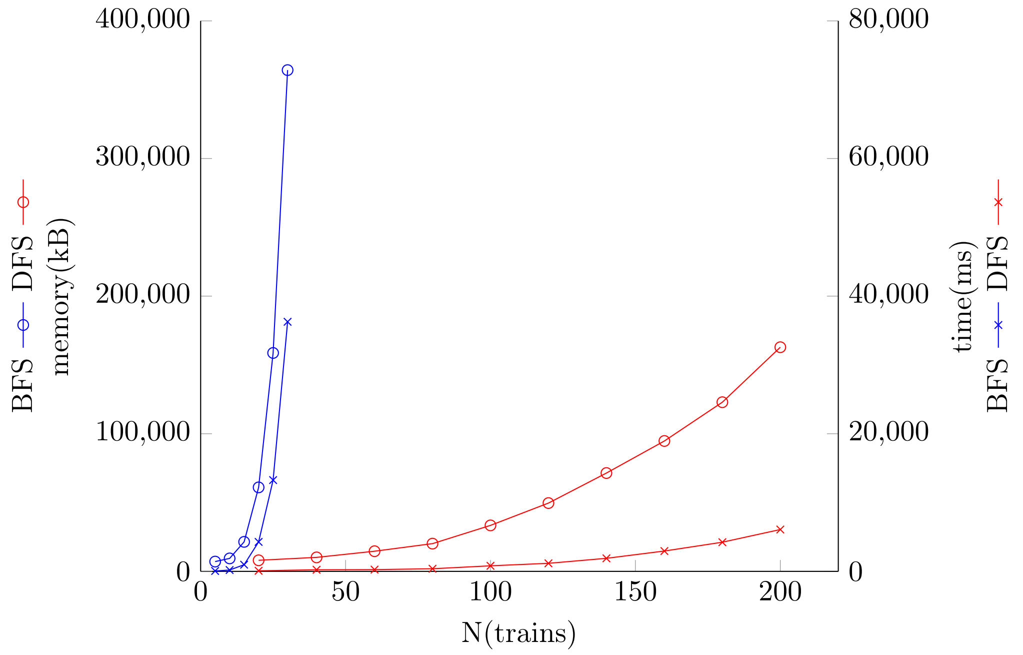


Figure 37.: Time/space performance of invariant violations (invariant violations reachable)

8.7.2. Diagnostic value

Similarly to deadlocks, invariant violations are best diagnosed by a trace, which is provided by the sanity check. However, it is not possible to provide a trace that ends in an invariant violation as such a state is undefined in UPPAAL. Therefore, the trace only points to the state before the invariant violation, leaving the user to deduct which transition points to the violation. Luckily, UPPAAL highlights transitions that violate invariants, which makes it easy for the user to diagnose the problem.
However, depending on the search strategy used, the diagnostic value of this trace is either very helpful for breadth-first search (as this guarantees a shortest trace), or potentially unhelpful for depth-first search (due to very long traces).
The usefulness of a very long trace depends on what information is needed to diagnose the cause of the violation: if only the last state is enough, then the size of the trace is not problematic. However, if the trace leading to the violation bears crucial information, then a very long trace will be hard to interpret.

8.7.3. Conclusion

Choosing between depth-first and breadth-first search bears a dilemma within: the user either has to wait very long for a helpful trace, or potentially be instantly provided by a very long trace.
As there is no right answer as to which is more helpful, we conclude that it is best to leave the search strategy as an option for the user.

8.8. Conclusion

We have done performance tests and case studies in order to evaluate the sanity checker and its internal implementations.
The performance tests have shown that using meta variables indeed increases the performance o the reachability checks. However, we have also seen that the second meta implementation did not behave as expected. We leave this for future work to investigate.
We have also applied the sanity checker to several UPPAAL models as case studies. While most of the models were proven correct by the sanity checker. A model for the Firewire protocol seemed mostly unreachable, which seems unintentional.

8.9. Case studies

In this section, we will apply all sanity checks to multiple UPPAAL and discuss the outcome of the sanity checks.
We use the following demo-models that are distributed with UPPAAL:

- `2doors`: two doors, each used by a person that must not be simultaneously open.
- `bridge`: four vikings are about to cross a damaged bridge in the middle of the night. The bridge can only carry two of the vikings at the time and to find the way over the bridge the vikings need to bring a torch simultaneously open.
- `fischer`: Fischer's mutual exclusion protocol.
- `scheduling3`: model of scheduler and tasks
- `scheduling4`: extended model of scheduler and tasks
- `firewire`: a case study on the Firewire protocol [18]

Also, we use a more complex model used for fault analysis of an electrically insulated railway joint [19]. We refer to this model as `joint`.

The results can be seen in Table 2.
Empty cells indicate that the sanity check did not detect violations. 'n/a' indicates that deadlocks could not be evaluated as on the model. All detections, labeled with a minus sign, could be subdivided into 3 categories. We discuss each category below:

1. Many unused variables were clocks that were used in queries in the XML file. However, the Ecore model discards these queries, which causes these clocks to

Model	SL	TL	SE	TE	SD	IV	UD
2doors							
bridge							- 1
fischer							
scheduling3	- 2	- 2	- 2	- 2			- 1
scheduling4	- 2	- 2	- 2	- 2			- 1
joint	- 3	- 3	- 3	- 3	n/a	- 4	- 1
firewire	- 5	- 5	- 5	- 5			- 1

Table 2.: Sanity check results of several UPPAAL models.
SL/TL=system/template location reachability, SE/TE=system/template edge reachability, SD=system deadlocks, IV=invariant violation, UD=unused declarations

become unused. This could be considered a soundness issue. Other unused declarations have usages somewhere in the model that are commented out.

2. The unreachable locations/edges in these categories are unreachable by design: they are error locations that are used to test the integrity of the model. Thus, the sanity checker can confirm that these locations are indeed unreachable.

3. The state space of the `joint` model is too big to be explored entirely. Therefore many potentially reachable locations appear unreached.

4. While verifying, UPPAAL reached a state whose successors are 'not well defined'. The diagnostic information pointed to an assignment assigning a data variable to a clock value, which is not allowed in verification. The `joint` model is meant to be used with UPPAAL's simulation engine. However, there are expressions that are allowed in simulation but not in verification, such as assigning data variables to clock values.

5. A very large portion of the Firewire-model is unreachable. It is, however, not clear whether this is intended.

9. Conclusions and future work

In this chapter, we will summarize the work we have done, look back at our research questions and discuss future work.

9.1. Conclusions

In this section, we will take our research questions and look back on how we answered them in this thesis.
First we will look at the three subquestions:

> **Subquestion 1** What are commonly made errors by users developing UPPAAL networks.

In Chapter 4, we looked at the application of sanity checks for UPPAAL, other transition systems and even programming languages. Based on our findings, many sanity checks for UPPAAL could be derived. Also, through personal communication with UPPAAL's users and developers, we have found more sanity checks, such as for invariant violations.

> **Subquestion 2** How can the selected errors be detected in a sound, complete, efficient and effective way.

In Chapter 7, we presented the actual implementations of 7 sanity checks. The sanity checks are designed with the four wanted qualities in mind: soundness, completeness, efficiency and effectiveness. We prove soundness and completeness in this chapter using the preliminaries on timed automata as presented in Chapter 3. Also in Chapter 7, we discuss the effectiveness of each implementation: what is the diagnostic value of the results of the sanity checks. In Chapter 8, we evaluate the efficiency of the implementations through performance testing. For sanity checks with multiple possible implementations, we compare implementations using the results of these test.

Subquestion 3 How can detected errors be properly communicated to the user

In Chapter 7, we also present the ways of communicating the results of the sanity checks to the user. Through UPPAAL's plugin system, we have access to the loaded model in the GUI. This enables us to use location/edge coloring to indicated reachability in the graphical editor, a very effective way of presenting location/edge reachability to the user. Other sanity checks like deadlocks and invariant violations provide traces which give important diagnostic value to the user in order to pinpoint the error. We are however limited, as we can not automatically load the trace without reverse engineering in UPPAAL's simulator. We are also still limited in giving feedback to the user in the declaration language.

With the subquestions answered, we can look at our main research question:

Research question How can a tool use sanity checks to help UPPAAL users find and correct command made modeling errors.

We have addressed the subquestions that are required to be answered in order to address the main question. In Chapter 6 we presented the main architecture that combines all components into a single tool. The tool is packaged as a plugin that can be added to the GUI of UPPAAL.

The end result is a sanity check built into UPPAAL that contains sanity checks that detect commonly made errors, and present helpful diagnostic information to the user.

Furthermore, we have applied the sanity checker to several models as a case study. For most models, no mistakes were detected, which increases the convince in the correctness of these models. However, for a model on the Firewire protocol, the sanity checker showed that many parts of the model were unreachable. Whether this is intentional or not, it shows how the sanity checker can quickly perform a sanity check for the user, which would have take hours if the user were to do this manually.

9.2. Future work

In this section we will give directions to possible future work on sanity checks for UPPAAL.

The following directions can be taken in future work:

- **Merge multiple sanity checks into a single UPPAAL verification** Currently, the worst case scenario is that for every sanity check, the whole state space of the model will be exhausted. This can be optimized by combining multiple sanity checks (and their transformations). For example, one could easily combine location or edge reachability with deadlock checking: first run the deadlock query, after which one could run the reachability query. This should be faster as the verifier can reuse the explored state space between verifications. It should also be possible to include invariant violation checking in this. This way, the whole state space only has to be calculated once, in stead of thrice.

- **Include more sanity checks for the declaration language** As we look at many popular programming languages, there are a lot of possible sanity checks that can be for the declaration language. Possibilities are: checking for dead code, expression simplification, enforce naming conventions, enforce indentations,

- **Sanity check feedback using syntax highlighting in the declaration language** Currently, it is not possible for an UPPAAL plugin to highlight code in the declaration language. In order to make the code editor feel more like a development environment instead of just an editor, the sanity check should be able to integrate more with the code editor.

Appendices

A. Transformation for invariant violations

```
Input: nstaOriginal : Ecore model
Output: nsta : transformed Ecore model
Begin
    NSTA nsta = EcoreUtil.copy(nstaOrig);
    DataVariableDeclaration dvd = new DataVariableDeclaration();
    dvd.setPrefix(DataVariablePrefix.META)
     dvd.setTypeDefinition(nsta.getInt());
    Variable var = new Variable("__isViolated__");
    dvd.getVariable().add(var);
    nsta.getGlobalDeclarations().getDeclaration().add(dvd);
    AssignmentExpression violate = new AssignmentExpression();
    IdentifierExpression id = new IdentifierExpression(var);
    violate.setFirstExpr(id);
    LiteralExpression lit = new LiteralExpression("1");
    violate.setSecondExpr(lit);
    ChannelVariableDeclaration cvdHigh = new ChannelDeclaration(nsta,
     "_high");
    cvdHigh.setBroadcast(true);
    nsta.getGlobalDeclarations().getDeclaration().add(cvdHigh);
    ChannelVariableDeclaration cvdHighest = new ChannelDeclaration(nsta,
     "_highest");
    cvdHighest.setBroadcast(true);
    nsta.getGlobalDeclarations().getDeclaration().add(cvdHighest);
    ChannelPriority cp = nsta.getGlobalDeclarations().getChannelPriority();
    ChannelList cpiHigh = GlobalFactory.eINSTANCE.createChannelList();
    ChannelList cpiHighest = GlobalFactory.eINSTANCE.createChannelList();
    cpiHigh.getChannelExpression().add(new Identifier(cvdHigh));
    cpiHighest.getChannelExpression().add(new Identifier(cvdHighest));
    cp.getItem().add(cpiHigh);
    cp.getItem().add(cpiHighest);
```

```
foreach Template t : nsta.getTemplates() do
    Location lBad = null;
    foreach Edge e : t.getEdges() do
        Location target = e.getTarget();
        if target.getInvariant() == null then
            continue;
        if lBad == null then
            lBad = new Location("_error_");
            lBad.setLocationTimeKind(LocationKind.COMMITED);
            t.getLocations().add(lBad);
        Location lCopy = new Location();
        lCopy.setLocationTimeKind(LocationKind.COMMITED);
        t.getLocations().add(lCopy);
        e.setTarget(lCopy);
        Edge eBad = new Edge();
        eBad.setSource(lCopy);
        eBad.setTarget(lBad);
        eBad.getUpdates().add(EcoreUtil.copy(violate));
        eBad.setSynchronization(cvdHigh, SynchronizationKind.SEND);
        t.getEdges().add(eBad);
        Edge eGood = new Edge();
        eGood.setSource(lCopy);
        eGood.setTarget(target);
        // invariantToGuard removes clock rate expressions
        eGood.setGuard(Util.invariantToGuard(target.getInvariant()));
        eGood.setSynchronization(cvdHighest, SynchronizationKind.SEND);
        t.getEdges().add(eGood);
return nsta;
```
Algorithm 3: Transforming Ecore model for invariant violations

B. Performance test results

	time(ms)		memory(kB)	
N(trains)	BFS	DFS	BFS	DFS
2	3	3	6,960	7,032
4	6	5	7,112	7,112
6	49	52	7,688	7,688
8	3,183	3,592	34,676	34,676
9	32,680	36,959	254,820	254,820

Table 3.: Time/space performance state exhaustion using breadth- and depth-first search (BFS/DFS resp.)

	time(ms)			memory(kB)		
N(trains)	naive	meta-1	meta-2	naive	meta-1	meta-2
10	105	72	72	10,392	7,396	7,196
20	163	65	61	23,740	8,164	7,940
30	414	105	72	73,824	9,452	9,072
40	795	154	85	181,188	11,336	10,436
50	1,532	222	115	372,876	14,352	12,440
60	2,711	277	170	750,644	18,484	16,176
70	4,735	352	223	1,540,384	22,880	20,100
80	7,243	475	302	2,192,900	28,764	24,728
90	11,619	617	422	3,426,628	35,528	27,060
100	NaN	805	581	NaN	41,844	37,672

Table 4.: Time/space performance of system location reachability (all locations reachable)

	time(ms)			memory(kB)		
N(trains)	naive	meta-1	meta-2	naive	meta-1	meta-2
2	130	101	89	6,952	7,048	7,060
4	72	76	88	7,156	7,148	7,156
6	163	153	241	13,952	7,736	7,380
8	5,122	4,845	9,709	463,988	44,348	34,868
9	49,905	48,338	95,789	3,830,972	346,388	258,244
10	NaN	NaN	NaN	NaN	NaN	NaN

Table 5.: Time/space performance of system location reachability (one location unreachable)

	time(ms)			memory(kB)		
N(trains)	naive	meta-1	meta-2	naive	meta-1	meta-2
20	42	54	55	9,416	7,588	7,572
40	119	132	127	14,304	9,776	9,724
60	196	219	209	25,656	13,584	13,552
80	280	417	414	62,120	20,832	20,688
100	681	728	708	74,264	32,068	29,504
120	1,005	1,174	1,159	155,540	47,980	47,692
140	1,449	1,872	1,865	197,292	69,592	66,712
160	2,095	2,840	2,834	224,628	83,564	81,620
180	3,994	4,132	4,140	452,960	111,120	127,860
200	5,130	5,946	5,936	631,532	147,532	165,652

Table 6.: Time/space performance of template location reachability (all locations reachable)

	time(ms)			memory(kB)		
N(trains)	naive	meta-1	meta-2	naive	meta-1	meta-2
2	32	33	22	6,924	6,980	7,048
4	36	35	54	6,992	7,172	7,072
6	153	113	197	14,060	7,688	7,648
8	4,971	4,749	9,278	464,080	40,740	34,188
9	51,581	51,555	96,410	3,830,488	311,780	254,232

Table 7.: Time/space performance of template location reachability (one location unreachable)

	time(ms)		memory(kB)	
N(trains)	meta-1	meta-2	meta-1	meta-2
2	46	50	7,052	7,052
4	48	38	7,040	7,040
6	98	116	7,748	7,748
8	4,663	4,585	43,260	43,260
9	45,797	44,963	339,176	339,176

Table 8.: Time/space performance of system edge reachability (all edges reachable)

	time(ms)		memory(kB)	
N(trains)	meta-1	meta-2	meta-1	meta-2
2	28	63	7,032	7,060
4	48	54	7,152	7,172
6	148	209	7,684	7,684
8	5,465	10,731	44,108	44,108
9	53,884	106,170	346,428	346,428

Table 9.: Time/space performance of system edge reachability (one edge unreachable)

	time(ms)		memory(kB)	
N(trains)	meta-1	meta-2	meta-1	meta-2
20	142	90	7,636	7,528
40	117	85	9,688	9,620
60	194	255	13,552	13,796
80	361	298	20,720	19,428
100	649	540	31,496	31,328
120	1,065	866	40,412	47,184
140	1,856	1,682	66,968	68,876
160	2,646	2,220	85,760	88,804
180	3,971	3,319	115,292	112,188
200	5,662	4,697	166,104	148,344

Table 10.: Time/space performance of template edge reachability (all edges reachable)

N(trains)	time(ms)	memory(kB)
2	79	7,028
4	57	7,028
6	180	8,076
8	6,783	52,524
9	67,154	420,284

Table 11.: Time/space performance of invariant violation (no violations reachable)

N(trains)	time(ms)	memory(kB)
5	58	7,120
10	195	9,424
15	970	21,444
20	4,283	61,032
25	13,266	158,708
30	36,271	364,252

Table 12.: Time/space performance of invariant violations (invariant violations reachable, breadth-first search)

N(trains)	time(ms)	memory(kB)
20	86	8,052
40	234	10,168
60	256	14,644
80	382	20,156
100	808	33,416
120	1,170	49,628
140	1,897	71,448
160	2,956	94,748
180	4,238	122,928
200	6,081	162,884

Table 13.: Time/space performance of invariant violations (invariant violations reachable, depth-first search)

Bibliography

[1] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183 – 235, 1994.

[2] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi, *UPPAAL — a tool suite for automatic verification of real-time systems*, pp. 232–243. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996.

[3] S. C. Johnson, "Lint, a c program checker," in *COMP. SCI. TECH. REP*, pp. 78–1273, 1978.

[4] G. Behrmann, A. David, and K. Larsen, "A tutorial on uppaal," *Formal methods for the design of real-time systems*, pp. 33–35, 2004.

[5] A. David and K. G. Larsen, "More features in uppaal," *Deliverable no.: D5. 12 Title of Deliverable: Industrial Handbook*, p. 49, 2011.

[6] "UPPAAL model API." http://people.cs.aau.dk/ marius/modeldoc/.

[7] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč, "Foundations of json schema," in *Proceedings of the 25th International Conference on World Wide Web*, WWW '16, (Republic and Canton of Geneva, Switzerland), pp. 263–273, International World Wide Web Conferences Steering Committee, 2016.

[8] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez, "Atl: a qvt-like transformation language," in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pp. 719–720, ACM, 2006.

[9] 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 *Dependable Software Engineering. Theories, Tools, and Applications* (K. G. Larsen, O. Sokolsky, and J. Wang, eds.), (Cham), pp. 319–336, Springer International Publishing, 2017.

[10] Aristotle, "Physics Book VI." http://classics.mit.edu/Aristotle/physics.6.vi.html. trans. by R. P. Hardie and R. K. Gaye.

[11] R. Gómez and H. Bowman, "Efficient detection of zeno runs in timed automata," in *Formal Modeling and Analysis of Timed Systems, 5th International Conference, FORMATS 2007, Salzburg, Austria, October 3-5, 2007, Proceedings*, pp. 195–210, 2007.

[12] J. Lind-Nielsen, H. R. Andersen, H. Hulgaard, G. Behrmann, K. Kristoffersen, and K. G. Larsen, "Verification of large state/event systems using compositionality and dependency analysis," *Formal Methods in System Design*, vol. 18, pp. 5–23, Jan 2001.

[13] G. H. Mealy, "A method for synthesizing sequential circuits," *Bell Labs Technical Journal*, vol. 34, no. 5, pp. 1045–1079, 1955.

[14] "visualSTATE user guide." ftp.iar.se/WWWfiles/vs/UserGuide.pdf.

[15] S. la Fleur, "Static analysis of symbolic transition systems with goose," 2018.

[16] "IntelliJ IDEA." http://www.jetbrains.com/idea/.

[17] E. Foundation, "Xtext." http://www.eclipse.org/Xtext/.

[18] A. David, K. G. Larsen, A. Legay, M. Mikučionis, and Z. Wang, "Time for statistical model checking of real-time systems," in *Computer Aided Verification* (G. Gopalakrishnan and S. Qadeer, eds.), (Berlin, Heidelberg), pp. 349–355, Springer Berlin Heidelberg, 2011.

[19] E. Ruijters, D. Guck, M. van Noort, and M. Stoelinga, "Reliability-centered maintenance of the electrically insulated railway joint via fault tree analysis: a practical experience report," in *Dependable Systems and Networks (DSN), 2016 46th Annual IEEE/IFIP International Conference on*, pp. 662–669, IEEE, 2016.