Diploma thesis

Enhancing UPPAAL's Explanatory Power using Static Zeno Run Analysis

Jonas Rinast

23. April 2012

1. Examiner: Prof. Dr. Sibylle Schupp
2. Examiner: Prof. Dr. Dieter Gollmann

Declaration

I, Jonas Rinast, solemnly declare that I have written this diploma thesis independently, and that I have not made use of any aid other than those acknowledged in this diploma thesis. Neither this diploma thesis nor any other similar work has been previously submitted to any examination board.

Hamburg, 23. April 2012

Jonas Rinast

Acknowledgments

I would like to thank my advisor Prof. Dr. Sibylle Schupp for the opportunity to work with her on an advanced research topic. I also thank her for her valuable input concerning my research and generally good supervision of my thesis development.

I am also very thankful for the support of my parents throughout my studies and for them enabling me to study at an university.

Abstract

This thesis discusses Zeno runs and their detection in timed automata networks. Specifically, we explore Zeno runs in the context of model checking with the tool UPPAAL. Zeno runs are transition sequences in the system that can execute arbitrarily fast, which conflicts with the real world experience where execution always takes time. Accordingly, most of the time the presence of Zeno runs in a model is unintentional. Also, Zeno runs can influence the model behavior in a negative way, for example timelocks, an analogy to deadlocks regarding time, may occur due to them, and therefore their detection is a desirable goal.

Gómez applied and extended Tripakis' strong non-Zenoness property concept to detect Zeno runs in timed automata systems. In this thesis we enhance Gómez' method by eliminating false positives that the approach yields in some circumstances. For this purpose we modify the way synchronization in the system is exploited by not only ensuring that valid synchronization partners exist but also that their amount is correct. Additionally, we incorporate two data variable heuristics because in most models certain Zeno runs can not occur due to constraints on data variables.

We implemented Gómez' algorithm and our extensions in an analysis tool named ZenoTool to validate the theoretical improvements in the detection accuracy due to our enhancements and also measured its run-time performance. We applied our tool to several real world case studies and the experiments show that the static analysis approach generally performs well. Positive evidence was found for our theoretical assumptions on the accuracy improvements and the tool's run time appears to be no issue. Thus, we conclude that static Zeno run detection can effectively contribute to ensure safety of timed automata network specifications and therefore enhances the explanatory power of model checkers like UPPAAL.

Contents

1. Introduction

Timed automata [3] are commonly used to model systems that need to adhere to timing constraints. They provide an easy-to-use, graphical way to specify components of the system. Several model checking tools are publicly available and can be used to verify the behavior of the system specification automatically. Concrete tools available are UPPAAL [5], which this thesis focuses on, Kronos [13], Red [37], and Profounder [35].

However, the specification of a timed automata system is not trivial and might incorporate flaws. One class of flaws are so called timelocks. They are extensions of deadlocks known from automata theory and are induced by the introduction of time, because the progress of the system might block due to timing constraints. Timelocks involving Zeno runs (Zeno-timelocks) can not be detected using simple deadlock detection algorithms. Zeno runs are paths in the timed automata system that execute infinitely many action transitions in finite time. Most of the time, such Zeno runs are not intentional and ensuring their absence is a desirable goal, because real systems can not execute arbitrarily fast. Model-checking tools like UPPAAL can verify liveness properties for this problem. However, the verification process needs to explore the whole state space of the timed automata system, which might render the verification of complex systems infeasible. This thesis therefore explores static analysis techniques to broaden the class of systems that can be proved safe in regards to Zeno runs. Together with deadlock freedom, the absence of Zeno runs then implies timelock freedom for the specification.

Tripakis presented a property for loops in a timed automata system called strong non-Zenoness (SNZ) [34], which characterizes the loop's possibility to add to a Zeno run. Based on this property, static and dynamic methods for dead- and timelock detection are proposed in the same paper. The SNZ property was then made applicable to a wider class of loops by Bowman and Gómez [12, 17]. They additionally proposed a check using invariants in the timed automata system that prevent Zeno-timelocks. In the case that those static analysis techniques are inconclusive, a hybrid method can be employed: using a set of reachability formulae derived from the specification, a sufficient-and-necessary condition for timelock absence can be constructed and verified by a model checker. Yet, Gómez applies his approach only to Zeno runs and not to Zeno-timelocks [22]. He proposes a method to construct an abstraction of the original specification that only contains loops prone to Zeno runs. This may reduce the state space of the timed automata system to a degree where liveness verification by a model checker might become feasible. Prevention of timelocks, in contrast to their detection, is also analyzed in the literature. For example, process algebras with asap [30] prevent timelocks due to mismatched blocking synchronization by limiting the use of urgency. Other proposals that also remove this class of timelocks by construction are Timed I/O Automata

[15], Discrete Timed Automata [20, 21], and Timed Automata with Deadlines (TAD) [7, 9, 8, 10]. Although UPPAAL cannot process Timed Automata with Deadlines directly, a transformation from a Timed Automata with Deadlines specification to a timed automata system for UPPAAL has been published by Gómez [18, 19] to complement their proposal.

This thesis improves the accuracy of a modern static Zeno run detection technique by refining the used synchronization exploitation. In addition, two heuristics are introduced that enhance the method by considering data variable valuations. The analysis presented in this thesis is largely based on the paper on Zeno run detection by Gómez [22]. At first, all simple loops in the timed automata system are extracted using a cycle detection algorithm. Existing algorithms [31, 32, 33] need to be adapted to process multiple edges between two states correctly as this is allowed in UPPAAL specifications but normal cycle detection algorithms assume at most one edge. Then the loops are checked for their strong non-Zenoness property by inspecting all the updates to clock variables and all the clock constraints on edges. Next, loops are declared safe or unsafe based on the SNZ property evaluation while taking into account that variables may be declared global in UPPAAL specifications and thus external updates to them may occur and invalidate the SNZ property. At this point, safe loops may be discarded because they can not contribute to any Zeno run in the timed automata system. The next step exploits the fact that the SNZ property is a compositional property: It is sufficient that only one of two synchronizing loops is strongly non-Zeno for the resulting loop in the product automaton to also be strongly non-Zeno. Gómez proposes the construction of a synchronization group to accommodate this fact. He finds the maximal set of unsafe loops such that all loops may synchronize on their respective channels with a partner also member of the set. In this thesis we propose another synchronization analysis, which yields more accurate results. Gómez' method may contain loops that cannot contribute to Zeno runs in practice as the maximal set approach overapproximates the synchronization scenario. We introduce a synchronization matrix, which assigns vectors to every unsafe loop depending on the channels the loop synchronizes on. By solving a linear (in)equation system for every loop we can then determine if there is a valid synchronization scenario involving the loop and thus dismiss loops without one. Gómez' last step reduces the initial UPPAAL specification to the set of unsafe loops found. This state space reduction may render feasible the verification of time divergence in the reduced specification using a model checker. This step is not included in this thesis, but could be incorporated at a later point in time as the step does not depend on the previous analysis steps. Concerning the data variable heuristics the first one extends the safety propagation to data variables such that loops depending on data variable valuations that can only be obtained from safe loop iterations are also safe. The second one eliminates loops that can not iterate at run time because of conflicting constraints that require different variable valuations.

We have implemented a tool called ZenoTool, which executes the analysis. Several models from different case studies were analyzed and the results of our method compared to the results obtained by Gómez' approach. The empirical results show that a static

analysis of an UPPAAL specification can efficiently ensure safety of a timed automata system regarding Zeno behavior. The experiments also validate the expectation that the new synchronization matrix approach improves the accuracy of the analysis, possibly lowering the load on subsequent analysis steps. Quantitatively, of the thirteen models analyzed three analysis results are more accurate than the results obtained by Gómez. One model profits from the enhanced safety propagation approach with synchronization matrices and the other two improvements are achieved by the data variable heuristics.

This thesis is divided into five chapters. Chapter 2 introduces the model checker UPPAAL and formally establishes the timed automata model used. Special characteristics of UPPAAL specifications are shown and UPPAAL's time model is depicted giving special attention to Zeno behavior. Chapter 3 presents methods to prove the absence of Zeno runs focusing on a static UPPAAL model analysis. Gómez' initial Zeno run detection algorithm is illustrated; based on it, the synchronization matrix construction is explained along with additional heuristics to eliminate false positives. Chapter 4 deals with the implementation of the presented analysis using C++. The underlying algorithms are given in more detail, the models and benchmarks, which were analyzed, are presented, and the analysis results are compared. Also, we point out supported features and limitations of the program. At last, chapter 5 reviews the contributions of this thesis and suggests further research topics related to the analysis presented.

2. UPPAAL

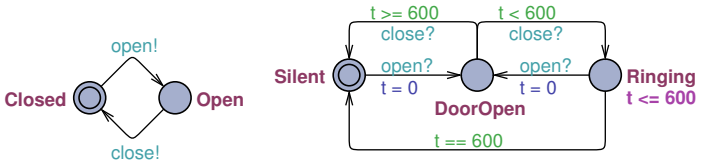
In 1995, the first version of UPPAAL, a toolbox for modeling and verifying timed systems, was released as a joint project of Uppsala University and Aalborg University [27]. It is based on the theory of timed automata [2] and allows verification of system properties using a subset of TCTL (timed computational tree logic) [1, 25]. Systems are modeled as a network of communicating finite state machines, which are subject to timing constraints. Up to today, steadily improvements are made to UPPAAL's verification engine. Some of the more recent additions in UPPAAL version 4.0 [6] are symmetry reduction [24], zone-based abstraction techniques [4], and user-defined functions. Several case studies used UPPAAL to successfully prove (or disprove) properties of, for example, communication protocols [11, 23, 28, 29] showing UPPAAL's relevance as a model-checker. For academic purposes UPPAAL is available free of charge at `http://www.uppaal.org`.

UPPAAL is divided into two separate applications: a graphical user interface (GUI) written in Java and the verification engine developed in C++. The verification engine can be used in a stand-alone fashion, but also interfaces with the GUI to allow easy access. The graphical user interface is split into three main parts. The editor allows the construction of timed automata systems. It provides a graphical representation of the finite state machines in the system, which can be edited easily. A C-like language is used for the declaration of variables, which can be used to annotate the automaton with constraints or to model data. The simulator then allows the user to execute transitions in the constructed system to test its functionality. However, the true power of the simulator lies within its feature to load error traces of the verification engine to provide diagnostic information to the user. In the verifier component of the GUI the user may specify properties for verification. These can then be checked directly from the component; upon failure, an error trace may be loaded into the simulator.

This chapter is structured in the following manner: Section 2.1 formally introduces a simplified timed automata model in UPPAAL. More elaborate modeling features not part of the formal model are presented as well. We also take a closer look at property verification using UPPAAL's query language. Section 2.2 revises UPPAAL's time model to clarify effects of timing constraint features. Special attention is given to features dealing with urgency. At last, section 2.3 introduces Zeno runs and related blocking states and gives formal definitions for them. Examples are provided to help the understanding of the concepts.

2.1. Timed Automata

The timed automata model used by UPPAAL is an extended model of the one proposed by Alur and Dill [3]. A system may consist of multiple, possibly concurrent components and every component is modeled as a finite state machine: a set of locations and edges with an initial location forming a directed graph. Communication between concurrent components is achieved by synchronization on self-defined channels and shared, bounded data variables. Timing constraints are realized by special clock variables, which take values in the positive reals: All clock variables advance simultaneously at the same rate but can be reset individually upon firing of an edge. Edges represent action transitions of the system. Basic annotations on edges are updates to variables including clocks (resets) and guards, i.e., basic boolean expressions on (time) variables. These define whether or not an edge may be fired at a certain state. Synchronization annotations are also possible. They declare that an edge may only be fired if a matching synchronizing second edge is available for firing. Time transitions are only possible in locations and are not represented by edges in the graph. This is why locations can be annotated with invariants. A state of the system may only stay in a certain location while the clock invariant is satisfied.



(a) Door model (b) Alarm device model

Figure 2.1.: Emergency door example model

Figure 2.1 shows an example UPPAAL model of an emergency door secured by an alarm system. As soon as the door is opened an alert signal will start to sound. The sound should stop if the door is closed and at least ten minutes have passed since its last opening. The model is divided into two parts. The door model (figure 2.1a) represents the physical door and has two locations for the open and closed states. Initially, the door is closed, which is shown by a second circle. Two edges model the open and close actions. They are annotated with synchronization labels `open!` and `close!`. The exclamation marks indicate that, when the edges are fired, a signal will be send on the channel with the same name and the alarm device will receive those to react accordingly. The alarm device model (figure 2.1b) is more complex: Initially it is in the `Silent` state and the only outgoing edge needs a matching `open!` signal for its receiving synchronization label `open?`. When the edge is fired the update `t = 0` is executed. Here, `t` is a clock variable that is used to measure the time passed since the door was last opened. The model then considers two cases when the door is closed again (synchronization of `close!/close?`). Either 600 time units have already elapsed (`t >= 600`) or they have not yet. If they have, the model just returns to the initial state and waits for the next opening of the

door. Otherwise the alarm sound needs to continue for the remaining time. The state `Ringing` models this behavior. The invariant `t <= 600` ensures that the alarm can not sound longer than ten minutes if the door was closed and the guard on the edge to the `Silent` state, `t == 600`, prevents an early switch off. Additionally, an edge is added to accommodate to the fact that of course the door can be opened again while the alarm is still ringing.

2.1.1. Additional Modeling Features

UPPAAL has a rich set of special constructs to enable users to specify more involved systems. In the following those will be presented in more detail.

Urgent locations, committed locations. In addition to simple locations one can declare locations to be urgent or committed. Both add implicit timing effects to the locations. In urgent locations, time is not allowed to pass. An enabled outgoing edge needs to be fired without delay. An urgent location is equivalent to a normal location that has an invariant using a new clock variable `new <= 0`. Additionally to the invariant, the new clock variable needs to be reset to zero on every incoming edge to the location.

Committed locations are extended urgent locations. Time is again not allowed to pass, but also the next action transition must involve a committed location. Other components may not fire non-committed action transitions until the system state no longer includes any committed location. Committed locations are useful for modeling atomic actions over multiple edges. For example, synchronization on multiple channels can only be achieved by using one edge per channel.

Broadcast channels, urgent channels. Binary synchronization channels, which match one outgoing signal (`channel!`) with one incoming signal (`channel?`), allow two different components' edges to fire simultaneously. In contrast, broadcast channels provide the user with a one-to-many synchronization construct. An edge annotated with a sending instruction on a broadcast channel (`broadcast!`) need not synchronize with any receivers. Sending on broadcast channels is always possible. However, if there are one or multiple components that have an enabled edge with a receiving action on the same channel (`broadcast?`), they will synchronize together and those edges will fire as well.

Urgent channels are used to force a synchronization as soon as it is possible. When a synchronization pair on an urgent channel becomes enabled it will fire without further delay. A limitation in UPPAAL regarding these features is that edges that synchronize on broadcast or urgent channels may not be annotated with timing constraints (using guards).

Data variables. UPPAAL provides its users with several types of variables, which may be part of the system state. Constant values may also be defined. Variables include simple bounded integers and booleans in the simplest case. They are updated like clock variables on edges. Guards also may involve data variables to decide if an edge is enabled. Standard arithmetic operations can be used to form complex expressions. A special case of variables are scalar sets. When using a scalar set, for example as a template

parameter, UPPAAL automatically applies symmetry reduction to the state space. A common scenario for variable usage is communication between different components of a system. To communicate, one component sets a variable and sends a synchronization signal. The second component then receives the signal and reads the variable to retrieve the value.

Arrays, records, custom types. Keeping the specification readable can easily be achieved by using UPPAAL's structuring features. Arrays of variables and record types over variables can be declared to aggregate related variables. Also custom types can be derived from existing ones in a C-like fashion. Such a custom type may be useful when, for example, declaring bounded integers (`int[0,5] var;`) as the upper and lower bound parameters are then encapsulated in the data type (`typedef int[0,5] bounded_t;`). When using scalar sets a declaration of a custom scalar set type is even mandatory for UPPAAL to apply symmetry reduction correctly.

Templates, parameters, system instantiation. In UPPAAL, system components are finite state machine templates, which may have parameters. Those parameters can be used to create multiple instances of the same automaton with different variable values or variable references. A special feature concerning these parameters is automatic parameter binding: When UPPAAL is instructed to instantiate a template and a parameter of that template was not bound to a concrete value beforehand, it will create one instance for every possible variable value. This behavior can for example be used to automatically create a desired amount of, e.g., sensor nodes by parameterizing a template with an identification number using the bounded integer type.

UPPAAL also allows partial parameter binding: One can derive a new template from a present one by binding some parameters to concrete values, furthering generic template construction. Concrete values in this sense may not only be variable values but also references to variables. This call-by-reference behavior enables the modeling of templates that, for example, synchronize on channels that are supplied by their parameters.

Selections, functions. In version 4.0 of UPPAAL two new features were introduced. Users may now define custom, C-like functions, which may be used in annotations of components. Updates and guard annotations of edges can refer to these functions and thus introduce more dynamic constraints. Also, a new convenient way to specify non-deterministic selection was added. Selection statements are annotations on edges. They are of the form `name : type`. A selection statement creates a variable `name` local to the edge, which is non-deterministically assigned a value from the range of `type`. This variable can then be used in guard, update, and synchronization expressions on the edge. An example usage is the non-deterministic selection of a specific channel from an array of channels to synchronize on. An equivalent specification can be obtained by adding individual, concrete edges for every possible valuation of `type`.

2.1.2. Modeling Syntax and Semantics

In the following, a simplified formalization of UPPAAL's timed automata model is given, which was also used by Gómez [22]. The formalization will neither incorporate ar-

ray, scalar, or record types nor functions, templates, or selection statements. However, bounded data variables (integers and booleans), all kinds of synchronization channels, and all location types are considered.

Basic definitions. Let $\mathcal{C}$ denote the set representing clock variables, $\mathcal{I}$ the set of bounded integer variables, and $\mathcal{B}$ the set of boolean variables. We denote the set of all variables $\mathcal{V} := \mathcal{C} \cup \mathcal{I} \cup \mathcal{B}$. Let $\mathbb{V}(\mathcal{V})$ be the set of all valuations of the variables in $\mathcal{V}$. Furthermore, we define the variable valuation function $\mathrm{v} \in \mathbb{V}(\mathcal{V})$, which maps a variable to its current value. More specifically, the function maps clock variables to the positive reals, bounded integer variables to the integers, which can be represented with n bits[1], and booleans to the discrete values 0 and 1.

$$\mathrm{v} : \begin{cases} \mathcal{C} \subseteq \mathcal{V} \to \mathbb{R}^{+0} \\ \mathcal{I} \subseteq \mathcal{V} \to \mathbb{Z} \setminus 2^n\mathbb{Z} \\ \mathcal{B} \subseteq \mathcal{V} \to \{0, 1\} \end{cases}$$

Assuming standard arithmetic, boolean and relational operators we then define $\mathbb{E}(\mathcal{V})$ to be the set of expressions involving the variables in $\mathcal{V}$. Furthermore, let $\mathrm{eval} : \mathbb{E}(\mathcal{V}) \to \mathbb{R}^{+0} \cup \mathbb{Z} \setminus 2^n\mathbb{Z} \cup \{0, 1\}$ be the expression evaluation function, which maps an expression, possibly containing variables, to its actual value by using the current variable valuation function. Lastly, we define the set of channel variables $Ch := BCh \cup UCh \cup BiCh$ consisting of broadcast channels BCh, urgent channels UCh, and standard binary channels $BiCh$.

Edge annotation syntax. Using the previous definitions we define UPPAAL's edge labels: At first, clock constraints are simple relational terms comparing a single clock or differences of clocks with an integer expression. Allowed relational operators vary by case and thus a generic set of relational operators $\mathcal{R}$ is assumed to define the set of clock constraints $\mathfrak{C}(\mathcal{R})$ involving one or two clocks x, y and an expression e:

$$\mathfrak{C}(\mathcal{R}) := \{ \quad x \sim e,\ x - y \sim e \quad | \quad x,\ y \in \mathcal{C},\ \ \sim \in \mathcal{R},\ e \in \mathbb{E}(\mathcal{V}),\ \mathrm{eval}(e) \in \mathcal{I} \quad \}$$

Guards are conjunctions of clock constraints $\mathfrak{C}(\mathcal{R})$ and boolean expressions without clocks. They allow any kind of relational operator on the clock constraints. Following this we define $\mathcal{R}_G := \{<, >, =, \geq, \leq\}$ and the set of guards $\mathfrak{G}(\mathcal{V})$ involving variables in the variable set $\mathcal{V}$.

$$\mathfrak{G}(\mathcal{V}) := \{ \quad \bigwedge_i e_i \wedge \bigwedge_j e_j \quad | \quad e_i \in \mathfrak{C}(\mathcal{R}_G),\ e_j \in \mathbb{E}(\mathcal{V} \setminus \mathcal{C}),\ \mathrm{eval}(e_j) \in \mathcal{B} \quad \}$$

The syntax of invariants is similar to that of guards but lower bounds on clocks are disallowed. Using $\mathcal{R}_I := \{<, =, \leq\}$ we define the set of invariants $\mathfrak{I}(\mathcal{V})$ involving variables in the variable set $\mathcal{V}$.

$$\mathfrak{I}(\mathcal{V}) := \{ \quad \bigwedge_i e_i \wedge \bigwedge_j e_j \quad | \quad e_i \in \mathfrak{C}(\mathcal{R}_I),\ e_j \in \mathbb{E}(\mathcal{V} \setminus \mathcal{C}),\ \mathrm{eval}(e_j) \in \mathcal{B} \quad \}$$

[1] UPPAAL uses 16-bit integers by default.

Note that in UPPAAL the conjunction operator is denoted by the symbol sequence `&&` or the keyword `and`. Next, Update annotations $\mathfrak{U}(\mathcal{V})$ in UPPAAL are comma-separated lists of individual assignments, which update the current variable valuation function. For clock variables, UPPAAL only allows assignments of non-negative integer values.

$$\mathcal{A}(\mathcal{V}) := var = e \quad \begin{cases} var \in \mathcal{V} \setminus \mathcal{C},\ e \in \mathbb{E}(\mathcal{V}) \\ var \in \mathcal{C},\ e \in \mathbb{E}(\mathcal{V}),\ \text{eval}(e) \in \mathcal{I} \geq 0 \end{cases}$$

Lastly, we define the set of possible synchronization labels.

$$\mathfrak{S}(Ch) := \{ \quad a?,\ a! \quad | \quad a \in Ch \quad \} \cup \{ \epsilon \}$$

Labels with a question mark indicate receiving actions and exclamation marks respectively indicate sending actions. The ϵ action is used for internal actions, which do not need synchronization. Note that UPPAAL does not allow one to annotate with clock constraints edges that synchronize on an urgent channel.

Timed automaton. A timed automaton is a tuple $A = (L, l_0, Lab, E, I, \mathcal{V})$. $L = ULocs \cup CLocs \cup NULocs$ denotes the set of locations with the respective subsets of urgent locations *ULocs*, committed locations *CLocs*, and non-urgent locations *NULocs*; $l_0 \in L$ is the initial location of the timed automaton and $Lab \subseteq \mathfrak{S}(Ch)$ the set of synchronization labels used by the automaton. $E \subseteq L \times Lab \times \mathfrak{G}(\mathcal{V}) \times \mathfrak{U}(\mathcal{V}) \times L$ is the set of edges and $I : NULocs \to \mathfrak{I}(\mathcal{V})$ a mapping function, which may assign an invariant to non-urgent locations. The last entry $\mathcal{V}$ is the set of variables the timed automaton uses. In this thesis, edges $(l, a, g, u, l') \in E$ will also be written in the format $l \xrightarrow{a,g,u} l'$, where a is the synchronization label, g the guard, and u the update.

Networks of timed automata. A network of timed automata is an aggregation of n single automata $|A = |\langle A_1, \ldots, A_n \rangle$, where $A_i = (L_i, l_{0,i}, Lab_i, E_i, I_i, \mathcal{V}_i)$. Variables with the same name used by multiple automata are global variables. Their set is given by $\bigcup_{1 \leq i \neq j \leq n} (\mathcal{V}_i \cap \mathcal{V}_j)$. Other variables are considered local to their components. A location in the network is represented by the location vector containing the positions in the individual components $\bar{l} = \langle l_1, \ldots, l_n \rangle,\ l_i \in L_i$. The notation used to indicate change of component locations is introduced next. The expression $\bar{l}[l_i'/l_i]$ refers to the location vector $\bar{l}$ that has its location l_i replaced by l_i'. Synchronous change to multiple components' locations uses a set notation $\bar{l}[(l_j'/l_j)_{j \in J}]$. This denotes the location vector $\bar{l}$, which results from replacing l_j by l_j' for every index $j \in J$. In analogy to the timed automaton definition, we also define $\mathcal{V} = \bigcup_{i=1}^{n} \mathcal{V}_i$, the set of all variables used by the network, and $I = \bigwedge_{i=1}^{n} I_i$, the invariant function for the network, which connects the invariants of the individual network components by conjunction, such that the function is applicable to the location vector. Lastly, $CLocs = \bigcup_{i=1}^{n} CLocs_i$, where $CLocs_i \subseteq L_i$ are local committed locations, refers to the set of committed locations in the network. The set of committed locations a certain location vector contains is denoted by $CLocs(\bar{l})$. In analogy, we define *ULocs* to refer to the set of urgent locations of the network and $ULocs(\bar{l})$ for the set of urgent locations in $\bar{l}$.

Variable semantics. Concerning updates to variable valuations we first formalize the update process on edges. If $u \in \mathfrak{U}(\mathcal{V})$ is an update sequence of the form $var_1 = e_1, \ldots, var_m = e_m$ and $\mathrm{v}_k \in \mathbb{V}(\mathcal{V})$ indicates the current variable valuation function, we define the resulting valuation functions, $\mathrm{v}_{k+i} \in \mathbb{V}(\mathcal{V})$, where $1 \leq i \leq m$:

$$\mathrm{v}_{k+i}(x) = \begin{cases} \mathrm{eval}(e_i) & x = var_i \\ \mathrm{v}_{k+i-1}(x) & \forall x \in \mathcal{V} \setminus \{var_i\} \end{cases}$$

The definition iteratively updates a single variable at a time by evaluating the corresponding expression using the current variable valuation function. Other variables remain constant. Using the intermediate variable valuation functions we define the final resulting variable valuation function $u(\mathrm{v}_k) = \mathrm{v}_{k+m}$.

Next, time progress regarding variable valuations is considered. Time delays are expressed by the operator $+ : \mathbb{V}(\mathcal{V}) \times \mathbb{R}^+ \to \mathbb{V}(\mathcal{V})$, which updates a variable valuation function v with a time delay δ.

$$(\mathrm{v}, \delta) \mapsto \mathrm{v}', \quad \mathrm{v}'(x) = \begin{cases} \mathrm{v}(x) + \delta & \forall x \in \mathcal{C} \\ \mathrm{v}(x) & \forall x \in \mathcal{I} \cup \mathcal{B} \end{cases}$$

Data variables are kept constant for any delay and all clock variables advance synchronously.

Automata network semantics. Using the symbol $\models$ to denote constraint satisfiability regarding variable valuations, the semantics of $|A$ can be given by a timed transition system $(S, s_0, \{\epsilon\} \cup \mathbb{R}^+, T)$, where $S \subseteq L \times \mathbb{V}(\mathcal{V})$ is the set of reachable states. We use $s = \langle \bar{l}, \mathrm{v} \rangle$ to refer to individual states and $s_0 = \langle \bar{l}_0, v_0 \rangle$, with $\bar{l}_0 = \langle l_{1,0}, \ldots, l_{n,0} \rangle$, and $\mathrm{v}_0(x) = 0, \forall x \in \mathcal{V}$, as the initial state. $T \subseteq S \times \{\epsilon\} \cup \mathbb{R}^+ \times S$ is the transition relation linking the source and destination states with a transition action. We consider two kinds of transitions: Action transitions are annotated with ϵ indicating that no time delay occurs. Delay transitions are annotated with $\delta \in \mathbb{R}^+$ referring to the occurring delay. Regarding the notation, we will use $s \overset{\epsilon}{\Longrightarrow} s'$ for action and $s \overset{\delta}{\Longrightarrow} s'$ for delay transitions, and $s \overset{\epsilon}{\Longrightarrow}$ for any action transition from s. In the following the five rules for transition computation are given:

1. Internal action transition (no synchronization)

$$\langle \bar{l}, \mathrm{v} \rangle \overset{\epsilon}{\Longrightarrow} \langle \bar{l}[l_i'/l_i], u_i(\mathrm{v}) \rangle$$

   Applicable for any transition $l_i \xrightarrow{\epsilon, g_i, u_i} l_i' \in E_i$ such that $\mathrm{v} \models g_i$, $u_i(\mathrm{v}) \models I(\bar{l}[l_i'/l_i])$, and $l_i \in CLocs_i$ or $CLocs(\bar{l}) = \emptyset$.

   This rule defines a single transition in a single component without any synchronization.

2. Internal delay transition

$$\langle \bar{l}, \mathrm{v} \rangle \overset{\delta}{\Longrightarrow} \langle \bar{l}, \mathrm{v} + \delta \rangle$$

Applicable for any $\delta \in \mathbb{R}^+$ such that $(\mathrm{v} + \delta) \models I(\bar{l})$, and $CLocs(\bar{l}) \cup ULocs(\bar{l}) = \emptyset$. Additionally no transition computation $\langle \bar{l}, \mathrm{v} + \delta' \rangle \stackrel{\epsilon}{\Longrightarrow}$ by using synchronization on urgent channels (see following rules) may be possible such that $\delta' < \delta$.

This rule defines a global delay transition. All clocks will advance by δ. However, time may not advance past a time where an urgent channel synchronization becomes available, as such a transition must be processed first.

3. Action transition using binary synchronization channel

$$\langle \bar{l}, \mathrm{v} \rangle \stackrel{\epsilon}{\Longrightarrow} \langle \bar{l}[l'_i/l_i, l'_j/l_j], u_j(u_i(\mathrm{v})) \rangle$$

Applicable for any transitions $l_i \xrightarrow{a!,g_i,u_i} l'_i \in E_i$ and $l_j \xrightarrow{a?,g_j,u_j} l'_j \in E_j$ $(i \neq j)$ such that $a \notin BCh$, $\mathrm{v} \models g_i \wedge g_j$, $u_j(u_i(\mathrm{v})) \models I(\bar{l}[l'_i/l_i, l'_j/l_j])$, and $\{l_i, l_j\} \cap CLocs(\bar{l}) \neq \emptyset$ or $CLocs(\bar{l}) = \emptyset$.

This rules defines a binary synchronization transition. Two edges annotated with matching synchronization labels $(a!/a?)$ in different components may be fired at once.

4. Action transition using broadcast synchronization channel (no receiver)

$$\langle \bar{l}, \mathrm{v} \rangle \stackrel{\epsilon}{\Longrightarrow} \langle \bar{l}[l'_i/l_i], u_i(\mathrm{v}) \rangle$$

Applicable for any transition $l_i \xrightarrow{b!,g_i,u_i} l'_i \in E_i$ such that $b \in BCh$, $\mathrm{v} \models g_i$, $u_i \models I(\bar{l}[l'_i/l_i])$ and there is no transition $l_j \xrightarrow{b?,g_j,u_j} l'_j \in E_j$ $(i \neq j)$ such that $\mathrm{v} \models g_j$, and $l_i \in CLocs_i$ or $CLocs(\bar{l}) = \emptyset$.

This rule defines a broadcast synchronization transition without any receiver. An edge sending on a broadcast channel may also fire, if there is no matching partner receiving on the channel. If there are receivers, the next rule applies.

5. Action transition using broadcast synchronization channel (with receiver(s))

$$\langle \bar{l}, \mathrm{v} \rangle \stackrel{\epsilon}{\Longrightarrow} \langle \bar{l}[l'_i/l_i, (l'_j/l_j)_{j \in J}], u_J(u_i(\mathrm{v})) \rangle$$

Applicable for any transition $l_i \xrightarrow{b!,g_i,u_i} l'_i \in E_i$ such that $b \in BCh$ and $J \subseteq [1..n] \setminus \{i\}$ is the maximal set of indices such that for any $j \in J$ there is a transition $l_j \xrightarrow{b?,g_j,u_j} l'_j \in E_j$, where $\mathrm{v} \models g_i \wedge \bigwedge_j g_j$, $u_J(u_i(\mathrm{v})) \models I(\bar{l}[l'_i/l_i, (l'_j/l_j)_{j \in J}])$, and $(\{l_i\} \cup \{\ l_j \mid j \in J\ \}) \cap CLocs(\bar{l}) \neq \emptyset$ or $CLocs(\bar{l}) = \emptyset$.

Note: Here u_J denotes the sequential execution of the updates $u_{j_0}, \ldots, u_{j_m}$ where $J = \{j_0, \ldots, j_m\}$ and $j_0 < j_1 < \cdots < j_m$. The order of update execution needs to be well-defined, because the expression evaluation in updates depends on the current variable valuation function, which changes after each individual update.

This rule defines a broadcast synchronization transition with receiver(s). If an edge sending on a broadcast channel is fired it will synchronize with all available receiving edges.

2.1.3. Verifying the System

Verification of properties of timed automata systems is the main goal of model-checkers like UPPAAL. Such properties need to be expressed in a formal way for the model-checker to process them. Several timed logics have been proposed in the literature. UPPAAL uses a simplified version of TCTL (timed computational tree logic) for its reasoning.

Timed computational tree logic is an extension to computational tree logic (CTL), which was introduced by Emerson and Clarke [14]. CTL is a specification language aimed at finite-state systems and originally allows formulae like $\exists\bigcirc\phi$, $\exists\phi_1 U\phi_2$ or $\forall\phi_1 U\phi_2$ where ϕ can either be a formula of this kind or an atomic proposition. The formulae enable the specifier to relate different states in regards to a proposition: for example the first formula $\exists \bigcirc \phi$ expresses that there exists a direct successor state in some state that satisfies ϕ. The other formulae express that either there exists a sequence of states, in which the property ϕ_1 holds in all states of the sequence except the last one, which instead satisfies ϕ_2, or all state sequences fulfill this requirement. Often the base set of formulae is extended with more convenient expressions: e.g. commonly $\exists\Diamond\phi$ expresses that a property ϕ will become true eventually (equivalent to $\exists$ true $U\phi$). However, considering such a formula, in CTL it is not possible to put a bound on when ϕ will become true. TCTL solves this problem by introducing time bounds on the operators. One can for example write $\exists\Diamond_{<5}\phi$ to express that ϕ will be satisfied within five time units in TCTL.

Two kinds of basic formulae are available in UPPAAL: state formulae and path formulae. In contrast to computational tree logic, path formulae may not be nested though. Path formulae can be further classified into three groups of formulae: reachability formulae, safety formulae, and liveness formulae. A more in-depth description of the classes is given in the following. Figure 2.2 gives an overview of UPPAAL's path formulae.

State formulae. A state formula is an expression that can be evaluated for a certain state without knowledge of the system behavior. For example, data variables could be checked to fulfill a certain property like $count < 5$, which would only be satisfied if the *count* variable is smaller than five. State formulae generally follow the syntax of guards, however, disjunctions are also possible. One can also check whether or not ϕ holds at a certain location l by using a formula of the form $P.l \implies \phi$. If one extends such a formula with path formulae, one can specify invariants that need to only hold in certain locations (see following paragraphs).

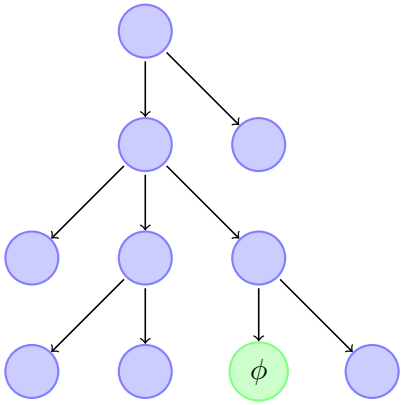
In UPPAAL, a special state formula for the detection of deadlocks is available, although technically it is not a state formula. The simple keyword `deadlock` represents this formula, which is satisfied in all deadlock states. A deadlock, according to UPPAAL, occurs if no outgoing edge is enabled in the current location and if no possible time delay will enable any of those edges ever again. In such a state leaving the location is impossible for all times. For technical reasons UPPAAL, currently only supports the usage of `deadlock` in simple safety and reachability formulae.

Reachability formulae. Reachability formulae are used to express whether or not a certain property ϕ can possibly become true in a reachable state. Most of the time,

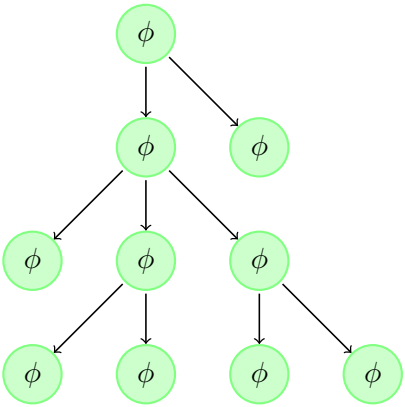
reachability formulae are used for sanity checks. A model might contain a certain error state and, using a reachability formula, one can prove that the specification is sound, in that it will never transition to that error state. Basic model behavior can be tested as well. For example, one can check if sending a packet in a communication protocol is possible at all. Distinguishing between reachability and liveness is important, as only sending a packet does not imply it will be received later on. An example of a satisfied reachability formula can be seen in figure 2.2a. From the initial state at the top, there is a path to a state where ϕ is satisfied. In UPPAAL a reachability formula $E\Diamond\ \phi$ is expressed by `E<>` ϕ.

Safety formulae. Safety formulae are used to express whether or not a certain property ϕ is invariantly true. Most of the time global properties of the system are ensured by them. For example, in a model of a nuclear power plant a reasonable safety formulae would restrict the nuclear reaction rate to a certain threshold at all times to ensure nuclear meltdown can not occur, whatever happens. A variation of the invariant nature of safety properties checks whether or not only a path in the reachable state space exists, where a certain property ϕ is satisfied in all states. The variation can be used to ensure at least one safe path exists. For example, in the case the model of the nuclear power plant can not invariantly guarantee a low reaction rate, because by improper control by a user the rate rises beyond all limits, one can at least verify that safe operation is possible by construction. Figure 2.2b and 2.2c respectively show possible state space models of satisfied safety properties. In figure 2.2b all states satisfy ϕ, in figure 2.2c one path satisfying ϕ exists. In UPPAAL the invariant safety properties $A\Box\ \phi$ and the potentially always safety property $E\Box\ \phi$ are expressed by `A[]` ϕ and `E[]` ϕ respectively.

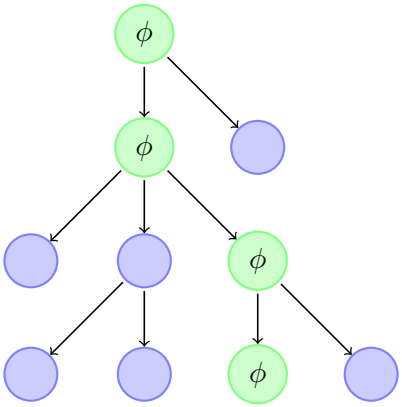
Liveness formulae. Liveness formulae are used to express whether or not a certain property ϕ will eventually become satisfied. A communication protocol might use liveness formulae to verify that packages sent eventually reach their destination. Also, the nuclear power plant model could ensure that the power plant eventually successfully powers down during an emergency. Two kinds of liveness formulae are available in UPPAAL: The simple one is the path formula $A\Diamond\ \phi$, which checks that the state formula ϕ will become true on all paths at least once. This can be used to show a model will eventually reach its goal from its initial state. The second formula is $\psi \rightsquigarrow \phi$, which is equivalent to $A\Box\ (\psi \implies A\Diamond\ \phi)$. It is satisfied if whenever ψ becomes satisfied, eventually ϕ will also become satisfied somewhere later along the path. Thus, one can verify timed reactions by using liveness formulae. Returning to the nuclear power plant, satisfaction of the formula *emergency* $\rightsquigarrow$ *powerdown* will show that the power plant is safe regarding emergencies: It will power down when an emergency occurs, no matter what. Figure 2.2d and figure 2.2e respectively show state space models of satisfied liveness formulae. In the first one the formula ϕ is satisfied eventually along all paths. In the second one ϕ only becomes satisfied after ψ was satisfied. UPPAAL uses `A<>` ϕ and ψ `-->` ϕ to express the liveness formulae.



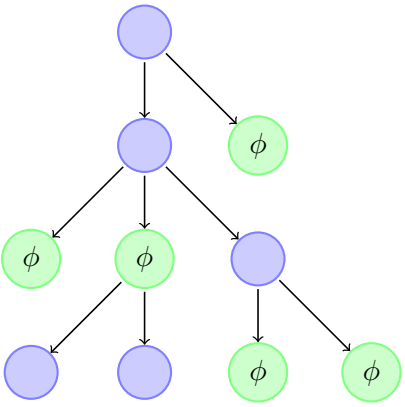
(a) Reachability formula: $E\Diamond\ \phi$



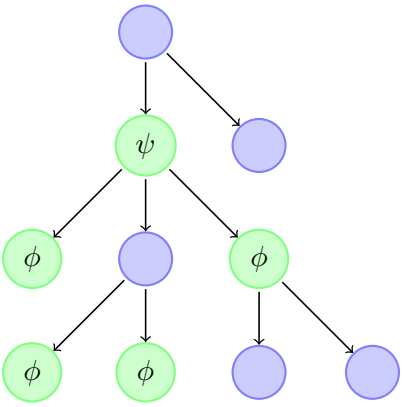
(b) Safety formula: $A\Box\ \phi$



(c) Safety formulae: $E\Box\ \phi$



(d) Liveness formula: $A\Diamond\ \phi$



(e) Liveness formula: $\psi\ \rightsquigarrow\ \phi$

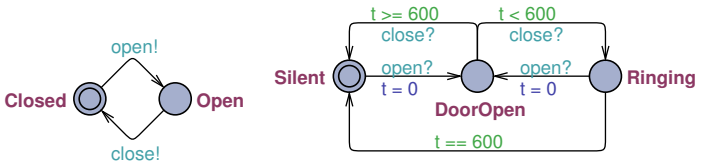
Figure 2.2.: Path formulae in UPPAAL, annotated nodes indicate satisfied properties

2.2. Time in Detail

To get a better understanding of UPPAAL's time model, this section revises the emergency door example given in section 2.1. A closer look at invariant and guard annotations together with satisfiable properties shows possible problems that may occur in specifications. Also, we present the semantics of urgency in more depth by extending the emergency door example.

2.2.1. Invariants and Guards

Distinguishing the influence of guards and invariants on the specification is important to obtain correct models. Invariants are used to impose upper bounds on clock variables in locations. Such invariants prohibit time to advance any further if the upper bound is reached. Time can only continue to progress if the clock is reset to a value less than the upper bound or the location is left to an unconstrained (or less constraint) location. In contrast to the must-behavior invariants exhibit, guards only impose a may-behavior. An edge is enabled if the current valuation of variables (including clocks) satisfies the guard. The edge, however, need not be fired. This behavior may lead to edges that, although they are enabled at one time, become disabled later on due to an upper time bound. False interpretation of the guard semantics may therefore lead to deadlock states or wrong model behavior.



(a) Door model (b) Alarm device model

Figure 2.3.: Modified emergency door model (no invariant)

By way of example, figure 2.3 shows a modified emergency door model. The invariant `t <= 600` in the state `Alarm.Ringing` was removed. Yet, the guard on the edge from `Alarm.Ringing` to `Alarm.Silent` still is only enabled after exactly 600 time units have elapsed. Closer examination of the state space using the verification engine yields that it now is possible for the alarm to sound even after 600 time units have passed; in difference to the unmodified model, the TCTL expression `A[] Alarm.Ringing imply Alarm.t <= 600` is no longer satisfied. In fact, if the transition to the `Alarm.Silent` location is not taken at 600 time units, the system will remain in the `Alarm.Ringing` location until the door is opened again. The alarm sound will not stop after 600 time units. A possible time run is given in figure 2.4. Correct behavior would transition to the `Silent` transition as soon as the value of `Alarm.t` hits the threshold of 600 time units.

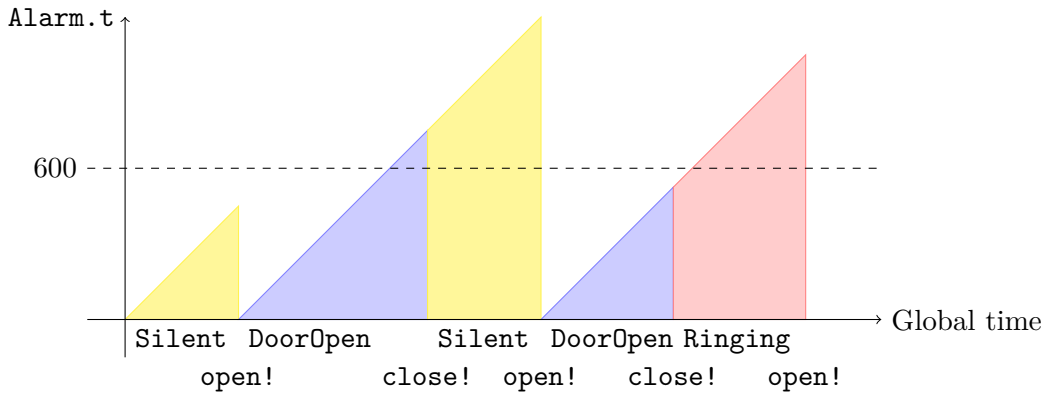
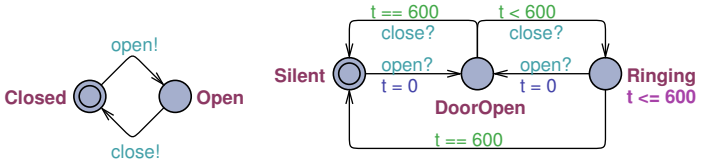


Figure 2.4.: A timing run of the emergency model without an invariant



(a) Door model (b) Alarm device model

Figure 2.5.: Modified emergency door model (invalid time space partitioning)

Another common mistake occurs because of false partitioning of time. Figure 2.5 shows the emergency model with another small modification. This time, the guard on the edge from the `Alarm.DoorOpen` to the `Alarm.Silent` location has been modified from `t >= 600` to `t == 600`. Now, the outgoing edges do not partition the reachable time space in the `Alarm.DoorOpen` location anymore. As time is unconstrained in the location and all ingoing edges reset `Alarm.t` to zero, the reachable time space is $\mathbb{R}^{+0}$. However, the outgoing edges do not cover $\mathbb{R}^{+0}$. The covered time $\mathcal{C} = \{\ t \mid t = 600\ \} \cup \{\ t \mid t < 600\ \} = \{\ t \mid t \leq 600\ \}$ is a subset of $\mathbb{R}^{+0}$ obtained by imposing an upper bound. Correct models do not need to cover the whole time space with outgoing edges in all locations. For example, a model might enable outgoing edges only after some time has passed. A partitioning case with an unenforced upper bound, though, introduces deadlock states to the model. Such deadlock states should be resolved by either enforcing the upper bound using an invariant or by employing a proper time space partition. In fact, for the modified emergency model (figure 2.5) the verifier fails to prove the TCTL expression `A[] not deadlock`, which implies a deadlock state. A possible path to reach the deadlock is visualized in figure 2.6, where after one successful iteration, the door is not closed early enough and a deadlock occurs. The initial example model given in 2.1 does not contain deadlock states.

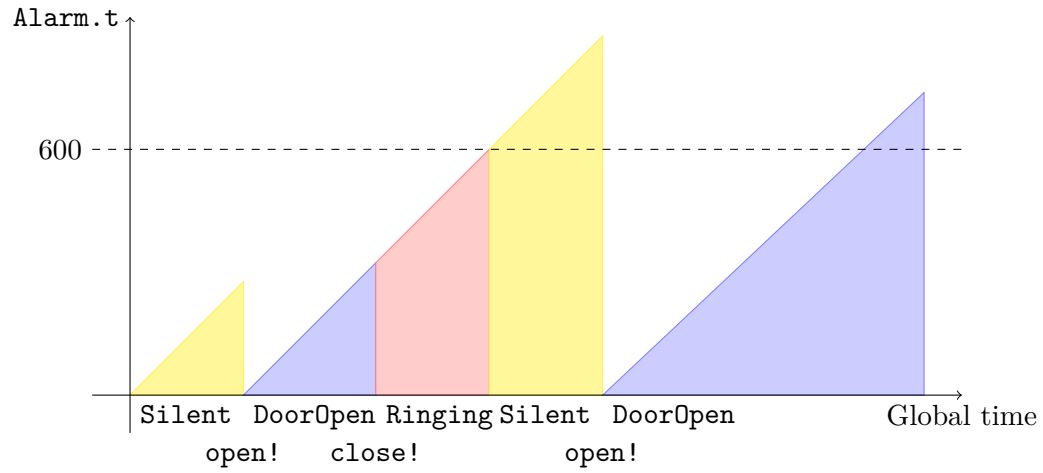
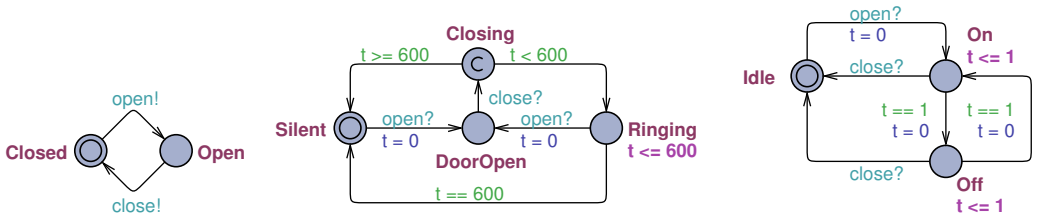


Figure 2.6.: A timing run of the emergency model with false time space partitioning

2.2.2. Urgency

UPPAAL features several mechanisms to specify urgent behavior. Urgency in this case means that a certain action may not be delayed any further and needs to be executed immediately. Using urgency correctly can be necessary to create models abiding to a certain specification. However, there are also specifications that do not need urgency, but benefit of its usage because the state space of the model might be reduced improving verification time.



(a) Door model (b) Alarm device model (c) Emergency light model

Figure 2.7.: Modified emergency door model

Returning to the emergency model example we change the specification to create a more complex model: in addition to the alarm sound, an emergency light shall blink while the door is open. Figure 2.7 shows the components of the new model. As now two components need to receive the `open!` and `close!` signals of the door, the corresponding channels were changed to broadcast channels. UPPAAL does not allow guards on edges that synchronize on broadcast channels. Therefore the alarming device specification (figure 2.7b) needs to be modified: the edges from `Alarm.DoorOpen` to `Alarm.Silent` and `Alarm.Ringing` have been replaced by two consecutive edges and the new interme-

diate location was marked as committed. In fact, only marking it urgent would have resulted in the same correct behavior. However, the usage of committed locations in contrast to urgent locations is superior because the state space of the model is then smaller as committed locations disallow interleaving with other system components before the committed state is left. The emergency light model depicted in figure 2.7c is straight-forward: Upon opening of the door, the system switches to the `Light.On` location. After one time unit imposed by an invariant and a guard it will transition to the `Light.Off` location. The same constraints apply here allowing the model to cycle between `Light.On` and `Light.Off` indefinitely many time units. Both states return to the `Light.Idle` location when the door closing signal (`close!`) is received. Figure 2.8 shows a timing run of the extended emergency model where the `Alarm.Closing` location is neither committed nor urgent. The resulting run does not comply with the specification: Although the door was closed after 600 time units the alarm device does not transition to the `Alarm.Silent` location immediately, resulting in a sound longer than intended. In fact, it is possible for the system to remain in the `Alarm.Closing` location forever. It follows that the door may never be opened again in that state.

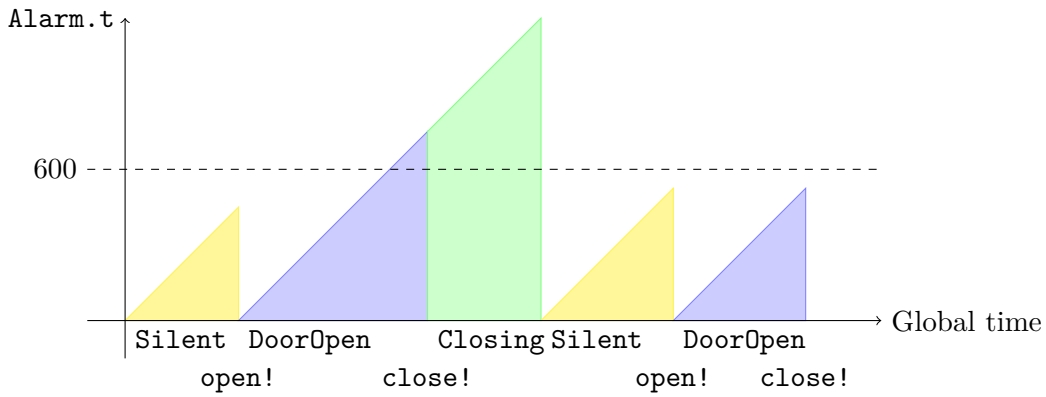
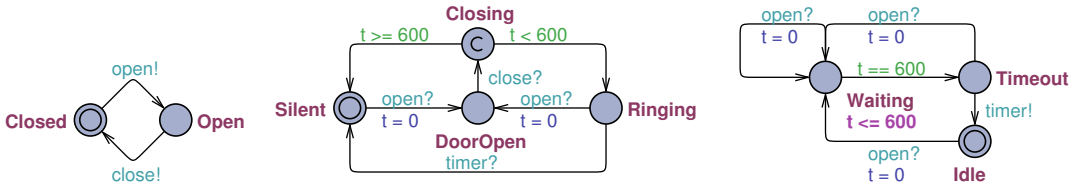


Figure 2.8.: A timing run of the extended emergency model without urgent behavior

At last we will demonstrate the use of urgent channels. Figure 2.9 shows again an extended modified model of the emergency door. This time, the alarm device model has been changed with regards to the timing constraints: The location `Alarm.Ringing` has no invariant anymore and the edge to the `Alarm.Silent` location is no longer guarded but now receives a synchronizing signal (`timer?`) on the urgent `timer` channel. As the transition to the `Alarm.Silent` location should happen exactly 600 time units after the door has been opened, a new `Timer` component (figure 2.9c) has been introduced to provide the urgent channel signal. Initially, the `Timer` component is idle and waits for the door to be opened. A local variable then makes the system wait in the `Timer.Waiting` state for 600 time units. An invariant (`t <= 600`) and a matching guard (`t == 600`) realize the waiting behavior. When the timeout is fired, the `Timer` component reaches the `Timer.Timeout` location. Here an edge with the urgent channel `timer` is present. Therefore whenever a matching synchronization partner becomes available they will

synchronize without further delay. By examination of the alarm device model one can now see that, if after 600 time units, the alarm device is not in the `Alarm.Ringing` location and thus capable of synchronizing, the door must not yet have been closed. This means the behavior of the whole system is correct, if one also considers that the door can be opened at any time by providing reset edges in all locations of the `Timer` component to initialize the timer again. Verification of the TCTL expression `A[] Alarm.Ringing imply Alarm.t <= 600` confirms that it is impossible to reach a state where the alarm sounds longer than 600 time units if the door was closed early.



(a) Door model (b) Alarm device model (c) Timer model

Figure 2.9.: Extended emergency door model with urgent channel

2.3. Timelocks and Zeno Runs

As seen in section 2.2 specifications of timed automata system models may incorporate flaws resulting from wrong usage of synchronization, false time constraints, or invalid application of urgency. One category of effects that can result from these flaws, are dead- and timelocks. This section will present and classify the different kinds of blocking states. In contrast to untimed systems, blocking can not only occur due to the inability to perform action transitions but also because time transitions may be unavailable.

Basic definitions. At first we define a run to be a path in the timed transition system established in section 2.1.2.

$$\rho := s_1 \overset{\gamma_1}{\Longrightarrow} s_2 \overset{\gamma_2}{\Longrightarrow} \ldots,\ s_i \in S, \gamma_i \in \{\epsilon\} \cup \mathbb{R}^+$$

A path ρ may be finite, ending in a state $s_n \in S$, or infinite. We use $Runs(s)$ to denote the set of runs starting at the state s. Furthermore, $FiniteRuns(s)$ ($FiniteRuns(s) \subseteq Runs(s)$) denotes the set of finite runs starting in s and we write $\rho \subseteq \rho'$ to denote that ρ' starts with the same state sequence as ρ. Next, we define the time divergence property of a run. A run ρ is time-divergent if the sum of all delays occurring in ρ is infinite.

$$\rho \text{ time-divergent} :\Longleftrightarrow \begin{cases} \sum_{i=0}^{\infty} \gamma_i = \infty, & \rho \text{ infinite} \\ \text{false}, & \rho \text{ finite} \end{cases}$$

Special attention needs to be given to so-called Zeno runs. A run ρ starting in state s is a Zeno run if

$$\neg\ (\rho \text{ time-divergent}) \wedge \rho \in Runs(s) \setminus FiniteRuns(s)$$

The definition characterizes an infinite path in the transition system where time converges. At a certain point, the path thus only consists of action transitions that are executed without delay between them. Infinite actions are executed in finite time. Obviously, Zeno runs are impossible in real systems and need to be closely analyzed to verify whether using a Zeno run approximation is acceptable or not. Absence of Zeno runs in a model is a good property for the model: deadlock freedom provided, timelock freedom is implied. An example of a Zeno run can be seen in the initial emergency door example (figure 2.1). The model allows opening and closing of the door without any delay. Thus, in theory, one could open and close the door infinitely often as at the same time the necessary synchronization is always possible. We use $\mathit{ZenoRuns}(s)$, which is a subset of $\mathit{Runs}(s)$, to denote the set of Zeno runs starting in state s.

Action- and timelocks. An actionlock is a state where no action transition can be performed even if an arbitrary number of delay transitions is taken. Formally, a state $s \in S$ is an actionlock if

$$\forall d \in \mathbb{R}^{+0}[(s+d) \in S \implies \nexists t \in T[(s+d) \stackrel{\epsilon}{\Longrightarrow}]]$$

where $s + d = \langle \bar{l}, \mathrm{v} + d \rangle$ if $s = \langle \bar{l}, \mathrm{v} \rangle$. In analogy, a timelock is a state in the timed transition system where all paths starting in the state are not time-divergent. In other words, as soon as the timelock state is reached, it is no longer possible for time to advance indefinitely, because the time is constrained by an upper bound on all possible paths. Formally, a state $s \in S$ is a timelock if

$$\forall \rho \in \mathit{Runs}(s)[\ \neg\ (\rho \text{ time-divergent})]$$

As time advances globally in UPPAAL's time model, a single clock entering a timelock will prevent time progress for the whole system thus blocking it completely. In the following, the different kinds of action- and timelocks are examined in more detail.

2.3.1. Pure-actionlock

Pure-actionlocks are equivalent to normal deadlocks in untimed transition systems. A state $s \in S$ is a pure-actionlock if

$$\forall d \in \mathbb{R}^{+0}[(s+d) \in S \wedge \nexists t \in T[(s+d) \stackrel{\epsilon}{\Longrightarrow}]]$$

An example of a pure-actionlock is given in figure 2.10. From the initial location, progress is only possible if the clock variable `t` takes a value greater than zero. Then, however, `t` is required to be zero for the other transition, which is not satisfiable. Time still can progress, though, and thus the system has reached a pure-actionlock.

2.3.2. Time-actionlock

Time-actionlocks do not only prevent actions from being executed but also time from progressing indefinitely. A state $s \in S$ is a time-actionlock if

$$\nexists t \in T[s \stackrel{\gamma}{\Longrightarrow},\ \gamma \in \{\ \epsilon\ \} \cup \mathbb{R}^{+0}]$$

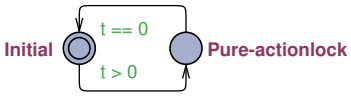
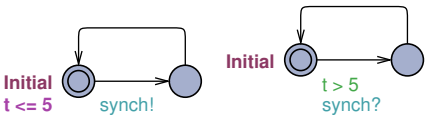


Figure 2.10.: A model with a pure-actionlock

An example of a time-actionlock resulting from invalid synchronization is given in figure 2.11. The sender component (figure 2.11a) synchronizes on the `synch` channel. However, time can at most be delayed until five time units have passed because of the invariant `t <= 5`. The receiver component (figure 2.11b) on the other hand can only synchronize on the `synch` channel after at least five time units have elapsed. Thus, the synchronization is never possible and both components remain in their initial states. Time can only be delayed for five time units, then the synchronization becomes urgent. Neither an action nor a delay transition is available, therefore a time-actionlock has been reached.



(a) Sender model (b) Receiver model

Figure 2.11.: A model with a time-actionlock

2.3.3. Zeno-timelock (Pure-timelock)

Zeno-timelocks or pure-timelocks only prevent time from progressing. Action transitions are still possible. A state $s \in S$ is a Zeno-timelock if

$$\forall \rho \in \mathit{Runs}(s)[\ \neg\ (\rho \text{ time-divergent}) \wedge \forall \rho' \in \mathit{FiniteRuns}(s)[\exists \rho'' \in \mathit{ZenoRuns}(s)[\rho' \subseteq \rho'']]]$$

All runs starting in s need to not be time-divergent. Also all finite runs need to be extensible to form a Zeno run, i.e. there needs to be a Zeno run that starts with the same state sequence of the finite run. This is necessary as a finite run that is not extensible leads to a time-actionlock instead of a Zeno-timelock, because the last state of the run has no action successor.

An example of a Zeno-timelock is given in figure 2.12. The only edge in the system can always be taken, thus action transitions are always possible. However, the invariant `t <= 5` prevents time from advancing beyond five time units. As soon as `t` reaches five time units, time can not progress any further but actions can be executed infinitely: a Zeno-timelock is present.

2.3.4. Property Concealment

As previously seen, the occurrence of Zeno runs may lead to Zeno-timelocks. Those themselves already pose a threat to the safety of the specification. However, Zeno

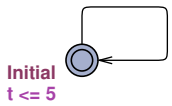
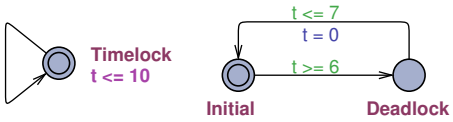


Figure 2.12.: A model with a Zeno-timelock



(a) Timelock model (b) Deadlock model

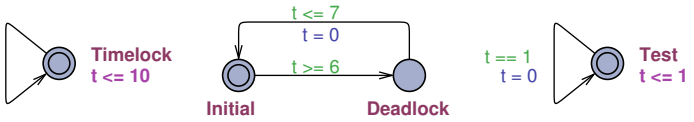
Figure 2.13.: A model that conceals a deadlock

runs may also exert influence on the specification in a different way: it is possible for Zeno runs to invalidate verification results. This happens because Zeno runs are always executable and thus deadlocks may always be circumvented by using the Zeno run transition. Also, Zeno-timelocks may restrict the reachable state space to a subspace that satisfies a certain property. However, in the complete state space the property may not be satisfied. As Zeno-timelocks can not occur in the real world the whole state space is reachable in reality. Such restriction of the state space is therefore unwanted, as it does not comply with the real world system. Liveness and deadlock properties are the most common properties to suffer from concealment.

Figure 2.13 shows a model that contains a concealed deadlock: If the time progresses further than $t = 7$, after the initial transition from `Concealed.Initial` to `Concealed.Deadlock`, a deadlock state has been reached. The guard `t <= 7` is disabled and there is no outgoing transition anymore. The whole system, however, may still fire the edge of the `Timelock` component and thus the property `A[] not deadlock` is satisfied. For emphasis: although the absence of deadlocks was confirmed, there still is an erroneous deadlock state in the model. This fact can be verified by removing the `Timelock` component and then rerunning the deadlock check. The deadlock freedom property will then be falsified.

3. Ensuring Time Divergence

Because of the negative influence Zeno runs exert on specifications, ensuring the divergence of time and thus the absence of Zeno runs in a specification is necessary to guarantee safety. In UPPAAL time divergence can not be verified directly, but indirectly: by using a test automaton, absence of Zeno timelocks (and Zeno runs) can be ensured. The test automaton consists of a single location with a single edge looping to itself. It uses a local clock and an invariant on the location together with a guard to ensure the only edge becomes urgent once every time unit. A loop that will iterate exactly once every time unit is the result. Now one can define a liveness property using the leads-to operator `-->`, which guarantees time divergence: $\lambda_U :=$ `Test.t == 0 --> Test.t == 1`. The formula λ_U checks if every iteration of the test automaton will eventually be followed by another iteration. Such an execution path only exists if no Zeno run (or Zeno-timelock) can occur. Figure 3.1 shows the concealed deadlock model from section 2.3.4 extended with the presented test automaton. The verification of the liveness formula λ_U indeed fails in this model.



(a) Timelock model (b) Deadlock model (c) Test automaton

Figure 3.1.: Concealed deadlock model with test automaton

Using λ_U, time divergence can be verified simply and robustly. But there are several disadvantages to the method as well. At first, liveness verification is computationally expensive and thus may not always be feasible depending on the complexity of the specification model. Verification of the formula λ_U is especially computationally expensive as the leads-to operator suffers from several limitations: the whole state space needs to be explored to guarantee absence of Zeno runs and thus on-the-fly verification is not possible. Also, symmetry reduction is not available for leads-to formulae eliminating an optimization method that can have a huge impact on verification speed.

Another disadvantage becomes obvious when examining the formula `A[] Test.t == 0 imply (A<> Test.t == 1)`, which is equivalent to λ_U. The equivalent formula shows that the verification of λ_U yields a conservative result: All paths following $t = 0$ need to satisfy $t = 1$ eventually. However, for ensuring absence of timelocks it is only necessary to find a single path that allows time to continue to progress. `A[] Test.t == 0 imply (E<> Test.t == 1)` would be the formula to verify the correct property. Unfortunately,

a matching formula is not expressible in UPPAAL and thus can not be verified.

Considering the disadvantages of indirect verification using UPPAAL, different approaches to ensure time divergence need to be developed to prove a wider class of specifications safe and to give more accurate results. Static analysis of the model specification can achieve these goals: the computational complexity of static analysis is low in contrast to dynamic analysis as the transition system is not executed. Verification of complex specifications may thus become feasible. Accuracy of the analysis is also increased by fine tuning the analysis and introducing sophisticated special case handling.

In this chapter a static analysis method for UPPAAL's timed automata model based on Gómez' results [22] is presented. Section 3.1 introduces the strong non-Zenoness (SNZ) property and defines the safety of loops. Section 3.2 then deals with propagation of the safety property by synchronization, broadening the set of safe loops. We will give two different approaches to the synchronization exploitation: the synchronization group method by Gómez (section 3.2.1) and a new method based on synchronization matrices (section 3.2.2). At last, section 3.3 will present two ways to incorporate data variable valuations into the analysis furthering the accuracy of the analysis.

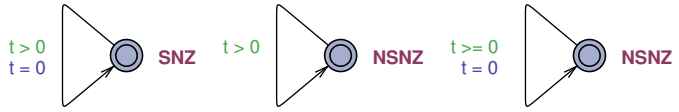
3.1. Strong Non-Zenoness and Loop Safety

Strong non-Zenoness (SNZ) is a static property for loops[1] in a timed transition system. SNZ was introduced by Tripakis [34] to classify loops according to their ability to contribute to Zeno runs. A loop is strongly non-Zeno if a clock variable t is bounded from below by a guard ($t \geq n, n \geq 1$) and the same clock is reset in the same loop ($t = 0$). This constellation ensures that time needs to pass when the loop is iterated. The guard only becomes enabled after a sufficient amount of time has passed as the clock is reset every iteration. If all loops in a network are strongly non-Zeno, the network is free from Zeno runs because, regardless ot the particular loop executed, time is required to advance. Such a network is also called strongly non-Zeno.

Bowman and Gómez modified the definition of strong non-Zenoness later on to allow more loops (and networks) to be classified accurately [12]. They restricted the amount of loops to analyze in the network to elementary cycles only. They also proposed the exploitation of synchronization, such that not all loops in a network need to be strongly non-Zeno to be free from Zeno runs.

Figure 3.2 shows some example loops and their classifications. In the loop in figure 3.2a the clock is reset to zero and bounded from below with a bound greater than zero. Therefore the loop is strongly non-Zeno. Figure 3.2b and 3.2c show examples of not strongly non-Zeno loops. The first loop misses a reset for its clock variable. Thus after an initial delay due to the guard the edge will always be enabled and Zeno runs may occur. The second NSNZ loop has a reset but the lower bound on the clock is not greater than zero. As a result the edge does not become disabled when the reset is executed

[1] Loops are cycles of action transitions in the automaton graph.

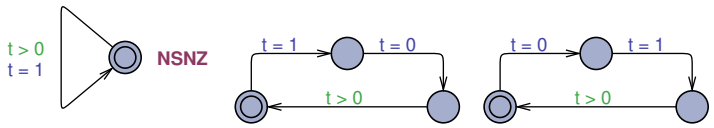


(a) SNZ model (b) NSNZ model #1 (c) NSNZ model #2

Figure 3.2.: Strongly non-Zeno and not strongly non-Zeno loops

and is thus always enabled. Again, Zeno runs may occur. In conclusion, the network consisting of all three loops is not strongly non-Zeno and may exhibit Zeno runs.

Unfortunately, the definition of strong non-Zenoness so far is not sound when taking into account advanced modeling features of UPPAAL like non-zero clock updates. A non-zero clock update refers to an update of the form `t = k` where $t \in \mathcal{C}$ and $k > 0$. Such updates may render lower bounds on clocks meaningless when inferring strong non-Zenoness. Even though a clock is bound from below and reset in a loop a Zeno run may occur if the valuation of the clock still satisfies the guard that imposes the lower bound after the reset has been executed. Another problem resulting from non-zero clock updates is that loops with multiple updates to the same clock rely on the order of update executions because later clock assignments may invalidate earlier ones.



(a) NSNZ model #1 (b) SNZ model (c) NSNZ model #2

Figure 3.3.: (N)SNZ loops involving non-zero clock updates

Figure 3.3 visualizes the special cases involving non-zero clock updates. In analogy to the previous examples (figure 3.2) the first example has a single loop. The clock is bound from below (`t > 0`) and reset (`t = 1`). However, the guard is still satisfied after execution of the update. Thus the loop is not strongly non-Zeno. Figure 3.3b and 3.3c depict the order problem of update executions. Both examples have two resets of the clock variable and one guard on it. The guard is satisfied if the clock variable is greater than zero (`t > 0`). The resets assign the values zero and one to the clock. Now, in the first example in figure 3.3b the reset to zero occurs last. Therefore the guard is not satisfied immediately and the loop is strongly non-Zero. In contrast, in figure 3.3c the reset to one is executed last. In this case the guard never disables the edge and Zeno runs are possible. Therefore the loop is not strongly non-Zeno.

Gómez resolves the issues of non-zero clock updates by refining the SNZ property even further [22].[2] The strong non-Zenoness property specialized on UPPAAL models

[2]He also improves propagation by synchronization. This is the topic of section 3.2.

is formalized in the following.

Loops. Let A be a timed automaton according to section 2.1. A *loop* in A is a transition sequence $\langle l_0 \xrightarrow{a_1,g_1,u_1} l_1 \dots l_{n-1} \xrightarrow{a_n,g_n,u_n} l_n \rangle$, where $l_0 = l_n$ and $l_i \neq l_j$ for all $0 \leq i \neq j \leq n$. Such a sequence is also called an elementary cycle of A. We distinguish two kinds of loops: *observable loops* and *internal loops*. *Observable loops* are loops that synchronize on a channel and thus can be observed from the outside. *Internal loops* are only composed of action transitions of the form $l_i \xrightarrow{\epsilon,g_i,u_i} l_{i+1}$ and accordingly do not synchronize in any way. Furthermore, a run *covers* a loop if all edges of the loop are fired infinitely often in the run.

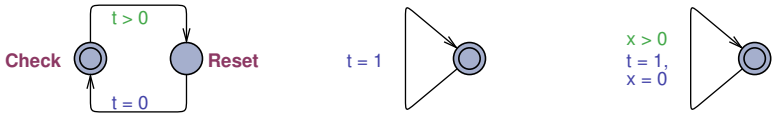
Strongly non-Zeno loops. Let Φ be a clock constraint and g a guard. $\Phi \in g$ then denotes that the constraint Φ can be inferred from the guard g. Next, let x and y be clocks, let m be a natural number including zero ($m \in \mathbb{N} \cup \{0\}$), and u be an update annotation on an edge. We use $x = m \in u$ to denote that the valuation of x is m after all assignments in the update u have been executed. In addition, if e is an edge in a loop lp annotated with a guard g and n is a natural number ($n \in \mathbb{N}$), we define $x_{\sqsupseteq n} \in g$ to express that either $x \sqsupseteq n \in g$ ($\sqsupseteq \in \{=, >, \geq\}$) or $x - y \sqsupseteq n \in g$ ($\sqsupseteq \in \{>, \geq\}$). Using those definitions we define x_{lb} to be the lower bound for x in g if $x_{\sqsupseteq x_{lb}} \in g$ and there is no $x'_{lb} > x_{lb}$ such that $x_{\sqsupseteq x'_{lb}} \in g$. At last, we use $\mathcal{U}(p, q)$ to denote the set of updates that occur on the edges on the path from p to q ($\langle p \xrightarrow{a_1,g_1,u_1} e_1 \dots e_{n-1} \xrightarrow{a_n,g_n,u_n} q \rangle$).

A loop lp is strongly non-Zeno (SNZ) if there exists a clock x with lower bound x_{lb}, an edge e_u with an update u, an edge e_g with a guard g in lp, and natural numbers m, m' that adhere to the following constraints.

$x = m \in u$	clock reset
$m < x_{lb}$	valid reset
$\forall u \in \mathcal{U}(e_u, e_g)[\nexists x = m' \in u[m' \geq x_{lb}]]$	lower bound invalidation

The first constraint ensures the clock variable x is reset in the loop. The second constraint assures time has to pass when iterating the loop by verifying the clock is reset to a value lower than the corresponding lower bound. At last, the third constraint guarantees that there is no update to the clock on the path from e_u to e_g, which invalidates the initial clock reset by assigning a value greater than the lower bound to the clock. A clock x that complies with those constraints is called an SNZ witness for the loop lp.

Unfortunately, in UPPAAL specifications a loop that is strongly non-Zeno is not guaranteed to prevent Zeno runs that cover that loop. The cause are external updates to non-local clock variables that may invalidate the assumption that time needs to pass for iterations: if a clock variable was not declared locally to a component but globally to all components, this clock may be an SNZ witness for a loop in a component. However, invalidating updates to that clock may occur in different components. As component actions may interleave to model concurrency it is possible for two components to interact in such a way that one component always invalidates the SNZ witness's clock reset and thus a Zeno run can occur.



(a) SNZ loop (b) Zeno update loop (c) Non-Zeno update loop

Figure 3.4.: Influence on SNZ by external updates

Figure 3.4 depicts a constellation of loops where a strongly non-Zeno loop is not safe with regards to Zeno loops. The loop in figure 3.4a is a simple, strongly non-Zeno loop: the clock variable t is reset to zero, there is a guard ensuring a lower bound greater than zero and no invalidating updates occur. However, if the clock variable t is not local to the component another component can update the value externally. We now assume that t is a global clock variable. Both figures 3.4b and 3.4c show loops that might occur in a different component. The first one is not a strongly non-Zeno loop and may exhibit Zeno runs. In addition, the global clock variable t is set to one every iteration. A possible run, which covers the SNZ loop, but, in spite of the SNZ property, exhibits Zeno runs, is any run that executes the loop in figure 3.4b first, then both edges of the SNZ loop and then repeats the execution in the same order. In such a run, t will be set to one, checked to be greater than zero and then set to zero again repeatedly. Thus, the run covers the SNZ loop. However, no time needs to pass and therefore such a run has no delays and it is a Zeno run. In contrast, the second loop in figure 3.4c also updates the global clock variable t to one, but is strongly non-Zeno itself (the local clock x is an SNZ witness). In this case time needs to pass, when the global clock is updated, and a Zeno run can not occur. If we now consider the complete model, the first loop is strongly non-Zeno but not safe from Zeno runs, as not all external updates to its SNZ witness clock t originate from safe loops. The second loop is not strongly non-Zeno and thus also not safe. And the third loop is strongly non-Zeno and also safe, because the loop possesses a local SNZ witness.

Safe loops. A loop lp in an UPPAAL specification is considered *safe* from Zeno runs if the following two conditions hold.[3]

1. lp is a strongly non-Zeno loop (SNZ)
2. lp has a local SNZ witness, or all external updates to an SNZ witness of lp originate from safe loops

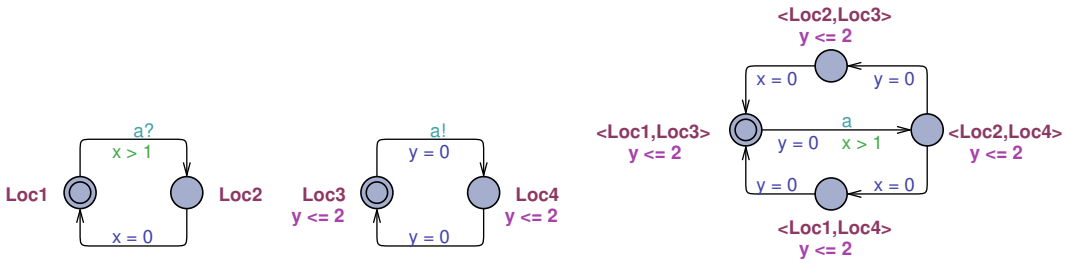
Following this definition a run that covers a safe loop will always be time-divergent and thus the safe loop can not contribute to the occurrence of Zeno runs: a run that covers a strongly non-Zeno loop is time-divergent by definition unless the SNZ witness clock is externally updated infinitely often without delays. Such updates require the loops, the

[3]Note: Gómez' definition does not require external updating loops to be safe, but to be SNZ with a local witness. Requiring them to be safe makes the safety property transitive.

updates originate from, to be loops that exhibit Zeno runs themselves. Such updates are impossible as a safe loop only receives updates from safe loops by definition and thus safe loops are free from Zeno runs.

3.2. Safety Propagation

Exploitation of synchronization between different components of a timed automata system may render it possible to deem a specification safe to Zeno runs even if unsafe loops exist in the specification. In fact, we are not interested in the safety of the individual components but rather in the safety of the product automaton that combines the individual components' automata into one big automaton.[4] The product automaton merges two edges that are subject to binary synchronization to a single edge. In addition, all clock constraints and resets are preserved. If two loops synchronize together it is therefore sufficient for one of both loops to be safe to obtain a safe loop in the product automaton. Safe specifications can thus be obtained without requiring all loops to be safe from Zeno runs.



(a) SNZ loop (b) NSNZ loops (c) Product automaton

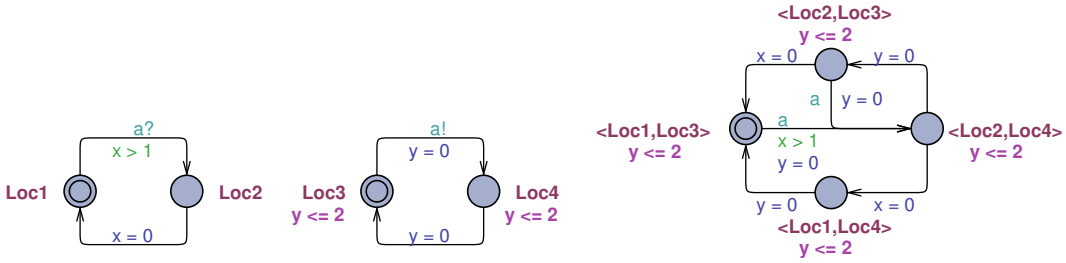
Figure 3.5.: Synchronization effects on (N)SNZ property

Figure 3.5 shows a system consisting of two concurrent loops that synchronize on a binary channel and the resulting product automaton. The first loop, depicted in figure 3.5a, is a simple strongly non-Zeno loop with a local SNZ witness x. The second loop, shown in figure 3.5b, is not a strongly non-Zeno loop. Invariants and clock resets on y were just added to show the effects on the product automaton (figure 3.5c). Two loops are present in the product automaton. They are created because the order of the transitions after the initial synchronization is not deterministic and both interleaving possibilities need to be represented: the loop via `<Loc2,Loc3>` is equivalent to a scenario where the not strongly non-Zeno loop returns to its initial location first. In analogy, the other loop is equivalent to the SNZ loop that transitions to its origin first. However, both loops have updates to x and constraints on x, such that they are considered strongly non-Zeno. Because the product automaton incorporates all concurrency of the original model no

[4] A formal definition of a product automaton can for example be found in Bowman's article on timelock detection [12].

external updates can occur and thus all SNZ loops are also safe loops. Therefore the product automaton only contains safe loops and is thus safe from Zeno runs, although it was constructed from partly unsafe components.

However, special attention needs to be given to synchronization on broadcast channels, because sending on a broadcast channel is non-blocking. Receivers need not be available for synchronization and still a sending loop can iterate. Thus, a synchronizing pair of a strongly non-Zeno loop and a not strongly non-Zeno loop may exhibit Zeno runs if the sending loop exhibits Zeno behavior. The problem is that in such a case synchronization is optional and not necessary for iterations and therefore the behavior of the sending loop is retained in the product automaton.



(a) SNZ loop (b) NSNZ loop (c) Product automaton

Figure 3.6.: Broadcast channel influence on synchronization

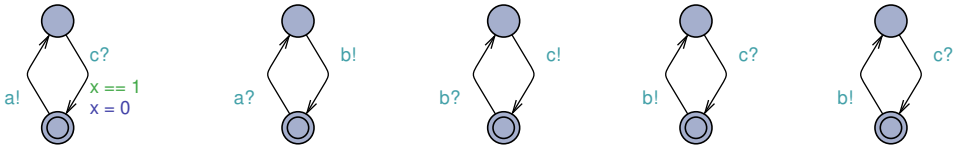
Figure 3.6 shows the same example system as figure 3.5, however this time the synchronization channel `a` is a broadcast channel. In this case the NSNZ loop may iterate even though the clock variable `x` is not greater than one. A synchronization partner edge is not necessary. The product automaton this constellation creates is given in figure 3.6c. In contrast to the previous product automaton an additional edge is present, which connects the locations `<Loc2,Loc3>` and `<Loc2,Loc4>`. This edge fulfills the case where the NSNZ loop transitions without a receiver. As a result a third loop is introduced to the product automaton, which just cycles between `<Loc2,Loc3>` and `<Loc2,Loc4>`. The loop is not strongly non-Zeno, although the product automaton was created from an SNZ loop and a NSNZ loop, conflicting with the result obtained from binary synchronization channels. In conclusion, techniques that exploit the propagation of loop safety need to treat binary and broadcast channels differently to accommodate the non-blocking behavior of a sending action on broadcast channels.

In the following, two such methods are presented. The first one (section 3.2.1) was proposed by Gómez [22] and yields a set of loops that may contribute to Zeno runs by ensuring valid, not strongly non-Zeno synchronization partners are present for all members of the set. The second method (section 3.2.2), which is a new proposal, also yields such a set. However, the method makes use of an (in)equation system to improve the accuracy of the analysis compared to Gómez' approach by eliminating certain impossible synchronization scenarios.

3.2.1. Synchronization Groups

The synchronization group approach finds a maximal set of unsafe loops, such that every loop in the set can synchronize on all channels it uses with a partner that is also member of the set. Groups of unsafe loops that can synchronize together are identified and therefore Zeno runs may occur. Maximality in this context refers to the amount of loops contained in the set. Thus all loops that fulfill the requirements need to be part of the set, otherwise loops exhibiting Zeno runs may be missed, when the specification is analyzed on the basis of the results of the synchronization group method. A formal definition is given next.

Synchronization group. Given a network of timed automata $|A$, UL_{sync} denotes the set of all unsafe, observable loops of $|A$. A synchronization group is a maximal, non-empty set $S \subseteq UL_{sync}$, such that, for any $lp \in S$ and any observable action in lp that synchronizes on a binary channel or receives on a broadcast channel, there is a matching action in some loop $lp' \in S$.[5] Synchronization groups will also be called *sync groups* in the following.



(a) Loop #1 (b) Loop #2 (c) Loop #3 (d) Loop #4 (e) Loop #5

Figure 3.7.: Synchronization group calculation

To get a better understanding on synchronization groups, figure 3.7 shows a system of timed automata that should be subject to the synchronization group analysis. All channels in the system are binary channels. Not considering strong non-Zenoness properties, there are three sets of loops, which can successfully synchronize together.

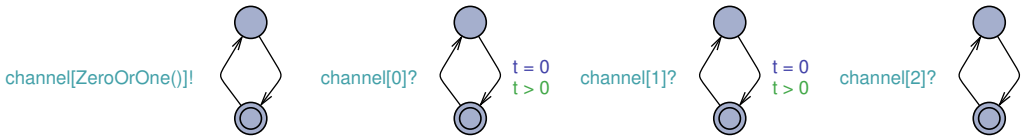
1. Loop #1, Loop #2, and Loop #3 (syncs on a, b, c)
2. Loop #3 and Loop #4 (syncs on b, c)
3. Loop #3 and Loop #5 (syncs on b, c)

Of the given loops only loop #1 is strongly non-Zeno and safe. Now, when applying the synchronization group analysis, only the unsafe loops are considered for grouping. Thus the first synchronization scenario is not possible anymore. Loop #1 would be required to be a member of the synchronization group so that Loop #3 has a synchronization partner for channel `c`. The second and third scenarios remain possible, because they consist entirely of unsafe loops. However, neither the set {Loop3, Loop4} nor the set

[5] Note: To our understanding Gómez' definition [22] is wrong as emitting broadcast actions do not need a synchronizing partner to successfully iterate, however receiving loops do.

{Loop3, Loop5} qualify as a synchronization group as the maximality criterion is not satisfied. The final synchronization group needs to include all loops of valid scenarios, therefore the synchronization group for the system is {Loop3, Loop4, Loop5}.

A problem regarding synchronization involves arrays of synchronization channels: if a synchronization channel is an array element, cases may occur where the analysis can only infer that synchronization should happen involving the array of channels, but the exact channel to synchronize on can not be identified. Such a case may occur if a synchronization label is of the form `channelarray[`e`]!/?` or an multidimensional variant of it and the expression e can not be resolved to a constant value; for example e could depend on data variables or selection statements, or involve function calls, whose values can not be derived statically. In such a case, the synchronization can not simply be omitted. Rather, one must ensure the synchronization that actually will happen will also be considered by the analysis. As a result, if a channel array element can not be identified, all elements of the channel array need to be considered valid partners for synchronization. This operation is conservative as the correct synchronization scenario will be captured, but impossible scenarios in the actual model may also be included.



(a) Loop #1 (b) Loop #2 (c) Loop #3 (d) Loop #4

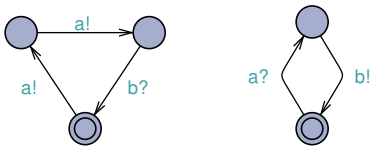
Figure 3.8.: Non-deterministic channel synchronization

The loop constellation given in figure 3.8 shows a model that uses an array of three synchronization channels. Imaging the function `ZeroOrOne()` to return zero or one non-deterministically. Thus, possible synchronization scenarios are only {Loop1, Loop2} and {Loop1, Loop3}. Both match an SNZ to a NSNZ loop on a binary channel. Therefore, the model is free from Zeno runs, if analyzed correctly. The synchronization group analysis can not infer the value of `ZeroOrOne()` and needs to assume all channels of the array may be used. This includes the actually impossible synchronization scenario {Loop1, Loop4}, which would exhibit Zeno runs if it was allowed. As a result the synchronization group constructed for the model is {Loop1, Loop4}, although such synchronization is impossible and the system is free from Zeno runs. A conservative overapproximation of the correct solution has been obtained.

3.2.2. Synchronization Matrix

The synchronization group analysis includes synchronizations that are impossible at runtime. For example, the order of synchronizations is not considered and also the amount of necessary matching partners is neglected. The model shown in figure 3.9 shows two such loops that can not synchronize together. Loop #1 sends on channel `a` twice and receives

on channel `b` once. Loop #2 only sends and receives the corresponding channels once. At run-time a deadlock will inevitably occur. In this case the synchronization group analysis will falsely return {Loop1, Loop2} as the sync group, because all synchronizations have matching partners.



(a) Loop #1 (b) Loop #2

Figure 3.9.: Impossible synchronization scenario

The synchronization matrix method improves the accuracy of the sync group analysis by eliminating such wrongly identified cases. At first, the method assigns all unsafe loops of the system a set of vectors representing the channels the loop synchronizes on. A set of constraint equations might be needed to link the vectors to obtain correct behavior. From the loop representation vectors a synchronization matrix is derived that models the synchronization possibilities of every loop. Then, by solving a constrained equation system valid synchronization scenarios involving certain loops can be found: a balance of sending and receiving synchronization labels needs to be found. If a balance involving a loop can be found the membership of a loop in the analysis result set can be implied.

At first we will focus on modeling the system correctly in section 3.2.2.1. Then, section 3.2.2.2 will present the synchronization matrix. Finding loops prone to Zeno runs by constructing a valid synchronization scenario will be explained in section 3.2.2.3. And at last, section 3.2.2.4 shows the improved accuracy of the method by applying the analysis to the model in figure 3.9 and a more complex example.

3.2.2.1. Loop Modeling

Let UL_{sync} refer to the set of observable, unsafe loops and let $Sync(S)$ be the set of synchronization channels used in the set of loops S. We denote the set of all synchronization channels used in unsafe, observable loops $\mathcal{L} = Sync(UL_{sync})$. In addition, we define $Vec(lp)$ to be the set of vectors assigned to the loop lp. $Vec(lp)$ is partitioned into two parts: the *base vectors* $Base(lp)$ and the *broadcast vectors* $Broad(lp)$:

$$Vec(lp) = Base(lp) \cup Broad(lp), \quad Base(lp) \cap Broad(lp) = \emptyset.$$

A vector $v \in Vec(lp)$ has one component per used synchronization channel and those components take values in the integral numbers ($v \in \mathbb{Z}^{|\mathcal{L}|}$). We refer to individual vector components by $x_{channel}$. The set of vectors assigned to a loop depends on the synchronization channels the loop uses. Again, we need to handle unresolved channel arrays in a special way. The argumentation for synchronization groups also applies in our case. In total four different channel classes need to be considered:

1. Binary channel (e.g., `chan channel`)
2. Broadcast channel (e.g., `broadcast chan channel`)
3. Unresolved binary channel array (e.g, `chan channel[2]`)
4. Unresolved broadcast channel array (e.g., `broadcast chan channel[2]`)

Vectors are assigned to loops in the following way: Initially for every $lp \in UL_{sync}$ the set $Vec(lp)$ consists of a single zero vector, denoting no synchronization at all. The initial vector is part of $Base(lp)$. We then iterate all loops $lp \in UL_{sync}$ and for every synchronization label lb of the form `c!`, `c?`, `a[`e`]!`, or `a[`e`]?` ($e \in \mathbb{E}(\mathcal{V})$) used in the loop lp the set $Vec(lp)$ is modified by the matching rule:

1. Synchronization on binary channel or receiving broadcast channel

   $$(\texttt{c} \notin BCh) \vee (lb = \texttt{c?})$$

   For every vector $v \in Base(lp)$, increase x_c by one, if lb is of the form `c!`, or decrease x_c by one, if lb is of the form `c?`.

2. Sending on broadcast channel

   $$\texttt{c} \in BCh \wedge lb = \texttt{c!}$$

   If not already in the set add a vector v to $Broad(lp)$, where x_c is set to one.

3. Synchronization on unresolved binary channel array or unresolved receiving broadcast channel array

   $$((\nexists \texttt{c} \in \texttt{a}[\texttt{c} \in BCh]) \vee (lb = \texttt{a[}e\texttt{]?})) \wedge e \text{ unresolved}$$

   For every channel `c` $\in$ `a`, create a copy $Copy_c$ of $Base(lp)$. Then, for every vector $v \in Copy_c$, increase x_c by one, if lb is of the form `a[`e`]!`, or decrease x_c by one, if lb is of the form `a[`e`]?`. The resulting base vector set is the union of all copies: $Base(lp) = \bigcup_c Copy_c$.

4. Sending on unresolved broadcast channel array

   $$\forall \texttt{c} \in \texttt{a}[\texttt{c} \in BCh] \wedge lb = \texttt{a[}e\texttt{]!}$$

   For every channel `c` $\in$ `a` if not already in the set add a vector v to $Broad(lp)$, where x_c is set to one.

Figure 3.10 shows one loop per rule. Assume the following channel variables: `a` is a binary channel, `b` is a broadcast channel, `c` is a binary channel array of size two, and `d` is a broadcast channel array of size two. All loops are unsafe and observable. We use the following form of vectors to represent the loops: $v = (x_a, x_b, x_c[0], x_c[1], x_d[0], x_d[1])^T$. The resulting loop models are:

(a) Loop #1 (b) Loop #2 (c) Loop #3 (d) Loop #4

Figure 3.10.: Synchronization cases for loop model generation

1. Loop: $Vec(lp) = \{\ (1,0,0,0,0,0)^T\ \} \cup \emptyset$
2. Loop: $Vec(lp) = \{\ (0,0,0,0,0,0)^T\ \} \cup \{\ (0,1,0,0,0,0)\ \}$
3. Loop: $Vec(lp) = \{\ (0,0,1,0,0,0)^T, (0,0,0,1,0,0)^T\ \} \cup \emptyset$
4. Loop: $Vec(lp) = \{\ (0,0,0,0,0,0)^T\ \} \cup \{\ (0,0,0,0,1,0)^T, (0,0,0,0,0,1)^T\ \}$

Note that the first set refers to the base vector set $Base(lp)$, while the second one is the broadcast vector set $Broad(lp)$.

3.2.2.2. Synchronization Matrix Construction

Based on the loop models we will now present the synchronization matrix, which will be used to calculate valid synchronization scenarios. The matrix has the following form:

$$\mathbf{S} = \begin{pmatrix} \mathbf{M} \\ \mathbf{C} \end{pmatrix} = \begin{pmatrix} \mathbf{M_1} & \mathbf{M_2} & \dots & \mathbf{M_n} \\ \mathbf{C_1} & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \mathbf{C_2} & \dots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \dots & \mathbf{C_n} \end{pmatrix}$$

Here $\mathbf{M_i}$ are model matrices for all unsafe, observable loops, and $\mathbf{C_i}$ are associated constraint matrices. Note that the constraint matrices $\mathbf{C_i}$ may be empty and thus blocks may be missing in a final synchronization matrix $\mathbf{S}$. The matrices $\mathbf{M_i}$ can be constructed from the loop's model by filling in the model vectors:

$$\mathbf{M_i} = \left(\vec{v_1} \dots \vec{v_n} \quad \vec{b_1} \dots \vec{b_n} \quad \vec{b_1} \dots \vec{b_n}\right), \vec{v_j} \in Base(lp_i), \vec{b_j} \in Broad(lp_i)$$

The constraint matrices $\mathbf{C_i}$ are used to relate the broadcast vectors to the base vectors. They are divided into three parts in the same way as the model matrices $\mathbf{M_i}$. They are of the form

$$\mathbf{C_i} = \begin{pmatrix} \mathbf{0} & s \cdot \mathbf{I} & -1 \cdot \mathbf{I} \\ \vec{l_i} \dots \vec{l_i} & \mathbf{L_i} & \mathbf{0} \end{pmatrix}$$

Here, $\mathbf{0}$ is the zero matrix, s is the total amount of synchronization labels[6], $\mathbf{I}$ is the identity matrix, $\vec{l_i}$ is the *base link vector*, and $\mathbf{L_i}$ is the *base link matrix*. There are

[6] A smaller bound could be established here by only considering receiving labels for the correct channel.

two kinds of constraints that are modeled in $\mathbf{C_i}$: *broadcast link constraints* and *base link constraints*. The upper blocks are used for broadcast link constraints and the lower blocks are used for base link constraints. Together the constraints model the one-to-many synchronization behavior of broadcast channels correctly. For the construction of the base link vector and matrix, we iterate through all synchronization labels lb in the loop lp_i and add values to $\vec{l_i}$ and rows to $\mathbf{L_i}$ if one of the following rules applies:

1. Sending on a broadcast channel

   $$lb = \texttt{c!} \wedge c \in BCh$$

   Add the amount of sending synchronization labels of the form `c!` in the loop lp_i to $\vec{l_i}$. Add a zero row to $\mathbf{L_i}$ and set the value x_p to minus one. The index p refers to the position of the vector $v \in \mathit{Broad}(lp_i)$, where $x_c = 1$. Note that we assume the set $\mathit{Broad}(lp)$ to be ordered in the same way as the vectors were added to the matrices $\mathbf{M_i}$.

2. Sending on unresolved broadcast channel array

   $$lb = \texttt{a[}e\texttt{]!} \wedge a[0] \in BCh$$

   Add the amount of sending synchronization labels of the form `a[`e`]!` in the loop lp_i to $\vec{l_i}$. Add a zero row to $\mathbf{L_i}$ and set the values $x_j, j \in P$ to minus one. The set P of indices refers to the positions of the vectors $v_j \in \mathit{Broad}(lp_i)$, where $x_c = 1$ and c is a channel in the array a. Again a matching order of vectors in $\mathit{Broad}(lp)$ is assumed.

Equation 3.1 shows the synchronization matrix $\mathbf{S}$ for the example system shown in figure 3.10. Vertical bars separate different loops and dashed vertical bars separate the base vectors from the broadcast vectors. The horizontal double bar separates the model matrices from the constraint matrices, the normal horizontal bars separate the loop constraint matrices, and the dashed horizontal bars separate broadcast link constraints from the base link constraints within one constraint matrix.

$$\mathbf{S} = \left(\begin{array}{cc:cc|cc:c:cccc}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
\hline\hline
0 & 0 & 4 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hdashline
0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 4 & 0 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 4 & 0 & -1 \\
\hdashline
0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & -1 & 0 & 0
\end{array}\right) \tag{3.1}$$

The first column models loop #1 and thus only sends on the channel `a` and has no associated constraints. The second loop is modeled by columns two to four. As it only sends on a broadcast channel the base vector set is unmodified and the only broadcast vector was duplicated. An associated broadcast link constraint links column three and four with values four[7] and minus one respectively. The base link constraint links the zero base vector in column two to the first broadcast channel in column three. The third loop that sends on an unresolved binary channel array with two elements is modeled in columns five and six. No constraints are necessary but the loop can either synchronize on `c[0]` or `c[1]` and thus two base vectors are created. The last loop sends only on an unresolved broadcast channel array. Accordingly the zero base vector is kept and for every channel in the array one broadcast vector were created and then duplicated. In the constraint section one can see the broadcast link constraints binding the duplicates together. The base link constraint in the last row models that per iteration of the loop only either `d[0]` or `d[1]` may synchronize.

3.2.2.3. Synchronization Scenario Calculation

The synchronization matrix S can now be used to calculate valid synchronization scenarios involving any loop. Thus loops that have no valid synchronization scenario, can be excluded from the list of loops that may cause Zeno runs. The (in)equation system that needs to be solved is given in equation 3.2.

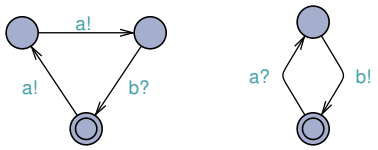
$$M\vec{x}^*_M = \vec{0}, \quad C\vec{x}^*_C \geq \vec{0}, \quad S = \begin{pmatrix} M \\ C \end{pmatrix}, \quad \vec{x}^* = \begin{pmatrix} \vec{x}^*_M \\ \vec{x}^*_C \end{pmatrix} \tag{3.2}$$

The equation system is further constrained, such that the components of the solution vector $\vec{x}^*$ may only take values in the natural numbers including zero ($\vec{x}^* \in (\mathbb{N} \cup \{0\})^n$). Upon successful solution calculation, the solution vector contains information about how often every loop needs to iterate to provide a matching synchronization partner for every synchronization label.

To decide whether or not a valid synchronization scenario exists for a certain loop lp, we solve the (in)equation system once for every base vector for the loop. Each time we require the component value of the solution vector $\vec{x}^*$ of a different base vector of lp to be greater than zero to ensure the solution will involve the loop. If any solution can be calculated, the loop may contribute to Zeno runs, otherwise it is eliminated from the result set of the analysis. The approach is correct as the synchronization matrix represents ingoing and outgoing synchronizations of loops and tries to find a balance by varying the amount of loop iterations. If no solution involving the loop lp can be found, there is no constellation of loop iterations where every synchronization has a valid synchronization partner. Thus, the system can not iterate when involving the loop lp and therefore lp need not be considered for Zeno run contribution.

[7] There are four synchronization labels in the model.

3.2.2.4. Accuracy Improvement



(a) Loop #1 (b) Loop #2

Figure 3.11.: Impossible synchronization scenario

Recall the example model from the beginning of section 3.2.2 that displayed the over-approximation of the synchronization group approach in figure 3.11. We now apply the synchronization matrix method to prove absence of Zeno runs in the system. At first, we model the two loops. Both loops are unsafe and observable. It follows that $UL_{sync} = \{Loop1, Loop2\}$, and $\mathcal{L} = \{a, b\}$. A vector v in the set $Vec(lp)$ will be of the form $(x_a, x_b)^T$. The loop in figure 3.11a sends twice on channel `a` and receives once on channel `b`. The loop in figure 3.11b only sends once on channel `b` and receives once on channel `a`. Because no broadcast channels are involved both loops are only assigned a single base vector:

1. Loop: $Vec(lp) = \{(2, -1)^T\} \cup \emptyset$
2. Loop: $Vec(lp) = \{(-1, 1)^T\} \cup \emptyset$

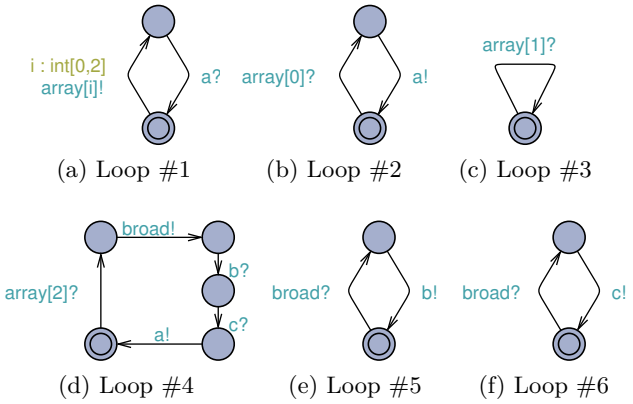
The construction of the synchronization matrix $\mathbf{S}$ is also simple, as no constraints for broadcast channels are created. The resulting equation system is

$$\left(\begin{array}{c|c} 2 & -1 \\ -1 & 1 \end{array}\right) \begin{pmatrix} x_a \\ x_b \end{pmatrix} = \vec{0}$$

We will solve the system once for every base vector for every loop. In this case, the equation system will be solved twice. On the first iteration we require x_a to be greater than zero. In analogy, on the second solving attempt we require x_b to be greater than zero. Both systems have no solution and thus the system is free from Zeno runs.

A more complex system is given in figure 3.12. The channel `broad` is a broadcast channel, all other channels are binary channels. Again, all loops are unsafe and observable. Thus, all loops are member of UL_{sync} and the set of used channels $\mathcal{L}$ is { `array[0]`, `array[1]`, `array[2]`, `a`, `b`, `c`, `broad` }. The loop modeling process creates the following sets if a modeling vector v is of the form $(x_{array[0]}, x_{array[1]}, x_{array[2]}, x_a, x_b, x_c, x_{broad})^T$:

1. Loop: $Vec(lp) = \{(1, 0, 0, -1, 0, 0, 0)^T, (0, 1, 0, -1, 0, 0, 0)^T, (0, 0, 1, -1, 0, 0, 0)^T\} \cup \emptyset$
2. Loop: $Vec(lp) = \{(-1, 0, 0, 1, 0, 0, 0)^T\} \cup \emptyset$
3. Loop: $Vec(lp) = \{(0, -1, 0, 0, 0, 0, 0)^T\} \cup \emptyset$



(a) Loop #1 (b) Loop #2 (c) Loop #3

(d) Loop #4 (e) Loop #5 (f) Loop #6

Figure 3.12.: Model involving complex synchronization

4. Loop: $Vec(lp) = \{(0,0,-1,1,-1,-1,0)^T\} \cup \{(0,0,0,0,0,0,1)^T\}$
5. Loop: $Vec(lp) = \{(0,0,0,0,1,0,-1)^T\} \cup \emptyset$
6. Loop: $Vec(lp) = \{(0,0,0,0,0,1,-1)^T\} \cup \emptyset$

The resulting synchronization matrix $\mathbf{S}$ is

$$\mathbf{S} = \left(\begin{array}{ccc|c|cc:cc|c|c} 1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ -1 & -1 & -1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & -1 & -1 \\ \hline\hline 0 & 0 & 0 & 0 & 0 & 0 & 14 & -1 & 0 & 0 \\ \hdashline 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 \end{array}\right)$$

If we consider the solution vector $\vec{x}^*$ to be of the form

$$(x_{Loop1,1}, x_{Loop1,2}, x_{Loop1,3}, x_{Loop2}, x_{Loop3}, x_{Loop4}, x_{Broad4,1}, x_{Broad4,2}, x_{Loop5}, x_{Loop6})^T$$

the equation system to solve is

$$\left(\begin{array}{ccc|c|c|c:c c|c|c} 1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ -1 & -1 & -1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & -1 & -1 \end{array}\right) \begin{pmatrix} x_{Loop1,1} \\ x_{Loop1,2} \\ x_{Loop1,3} \\ x_{Loop2} \\ x_{Loop3} \\ x_{Loop4} \\ x_{Broad4,1} \\ x_{Broad4,2} \\ x_{Loop5} \\ x_{Loop6} \end{pmatrix} = \vec{0}$$

under the constraints

$$\left(\begin{array}{ccc|c|c|c:c c|c|c} 0 & 0 & 0 & 0 & 0 & 0 & 14 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 \end{array}\right) \begin{pmatrix} x_{Loop1,1} \\ x_{Loop1,2} \\ x_{Loop1,3} \\ x_{Loop2} \\ x_{Loop3} \\ x_{Loop4} \\ x_{Broad1} \\ x_{Broad2} \\ x_{Loop5} \\ x_{Loop6} \end{pmatrix} \geq \vec{0}$$

Solving the system while always requiring one of the x_{Loop} components to be greater than one yields, for example, the following results:

1. $x_{Loop1,1} > 0 : x^* = (1,0,0,1,0,0,0,0,0,0)^T$
2. $x_{Loop1,2} > 0$: No solution
3. $x_{Loop1,3} > 0 : x^* = (0,0,1,0,0,1,1,1,1,1)^T$
4. $x_{Loop2} > 0 : x^* = (1,0,0,1,0,0,0,0,0,0)^T$
5. $x_{Loop3} > 0$: No solution
6. $x_{Loop4} > 0 : x^* = (0,0,1,0,0,1,1,1,1,1)^T$
7. $x_{Loop5} > 0 : x^* = (0,0,1,0,0,1,1,1,1,1)^T$
8. $x_{Loop6} > 0 : x^* = (0,0,1,0,0,1,1,1,1,1)^T$

Evaluating these results, the third loop can be excluded from the list of loops that can contribute to Zeno runs. All other loops have at least one valid synchronization scenario involving them and thus may yield Zeno runs. The final result set the synchronization matrix method creates is { Loop1, Loop2, Loop4, Loop5, Loop6 }. In contrast the synchronization group approach yields { Loop1, Loop2, Loop3, Loop4, Loop5, Loop6 }.

3.3. Data Variable Heuristics

Data variables play a huge role during execution of a timed transition system in UPPAAL but have not yet been analyzed regarding their influence on Zeno runs. They may, for example, disable edges or dynamically provide upper and lower bounds for time constraints and thus modify the behavior of a model significantly. Obviously, during a static analysis exact values for data variables can not be inferred directly. However, techniques using data flow analysis can restrict the set of possible valuations for a data variable at certain states. Such restrictions may render possible the elimination of loops that can not iterate due to data variable constraints and thus improving the accuracy of the Zeno run analysis even further.

In this thesis we do not apply a fully-fledged data flow analysis to UPPAAL's timed automata system. Instead, we decided on an ad-hoc approach that evaluates loop constellations and variable valuations on a case by case basis to eliminate impossible Zeno runs. The next two subsections will present two such heuristics to remove false positives from the result set of the synchronization matrix analysis. In section 3.3.1 we present a method dealing with unsafe loops that have a data variable constraint that can only be satisfied by executing safe loops. Thus, such a loop can also be considered safe. Section 3.3.2 then describes a heuristic that eliminates loops that have two or more guards that require conflicting variable valuations for their satisfaction.

3.3.1. Safe Variable Dependencies

UPPAAL's specifications may contain loops that are unsafe when only considering clock variables but safe when also including data variable valuations. Using data variable dependencies it is possible to remove loops, which were falsely deemed vulnerable to Zeno runs.

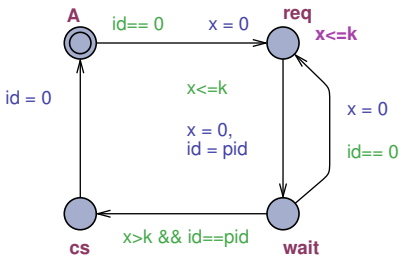


Figure 3.13.: Fischer protocol automaton

Consider the timed automaton given in figure 3.13. The automaton consists of a single component that models the Fischer synchronization protocol [26]. The component uses two variables and two constants: `x` is a local clock variable, `id` is a global integer variable, `pid` is a unique, constant integer value that is greater than zero to identify the component, and `k` is a global, constant integer value greater than zero. The automaton has two loops:

1. Loop: `A` → `req` → `wait` → `cs` → `A`

2. Loop: `req` → `wait` → `req`

The first loop is a safe loop as `x` is reset on the edge `A` → `req` (`x = 0`), `x` has a lower bound on the edge `wait` → `cs` (`x > k`), and `x` is a local clock. The second loop is neither safe nor strongly non-Zeno. However, the second loop has a safe data variable dependency: the data variable `id` is required to be zero (`id == 0`) for the loop to iterate. In addition, the loop itself sets the value of `id` to a value that does not satisfy the guard (`id = pid`) and thus the loop can not iterate on its own indefinitely. Instead, the loop depends on external influence on the data variable `id`. In this example, a valuation of `id` that satisfies the guard is only reachable if the first, safe loop is executed. Therefore, the second loop can also be considered safe because iterations of the second loop always require a safe loop iteration of the first loop ensuring time divergence.

Safely dependent loops. Let lp be an loop in a network of timed automata $|A$. A loop is *safely dependent* if there exists an edge e_u with an update u and an edge e_g with a guard g in lp, such that $\forall\langle\bar{l}, \mathrm{v}\rangle \in S[u(\mathrm{v}) \not\models g]$, and every reachable variable valuation v' such that $\mathrm{v}' \models g$ originates from an update u' on an edge that is part of safe loops only.

Following this definition every run that covers a *safely dependent* loop lp is time-divergent, because such a run needs to cover not only the loop lp but also at least another loop lp' that provides a variable valuation v' to satisfy g. As all loops that provide such a valuation are safe by definition, the run covers a safe loop and thus the run is time-divergent.

3.3.2. Conflicting Guards Elimination

When specifying a model in UPPAAL the user may create loops in the system that can not execute during model execution due to constraints. However, those loops are still considered for Zeno run detection during a static analysis run. The conflicting guards elimination approach removes a subset of such loops by finding guards in a loop that require conflicting variable valuations to be satisfied.

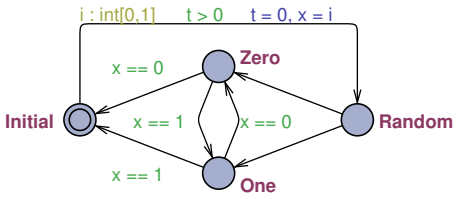


Figure 3.14.: Model with an impossible loop

The model in figure 3.14 shows a model with such an impossible loop. The model uses two local variables; `x` is an integer variable and `t` is a clock variable. Five loops are present in the system:

1. Loop: `Initial` → `Random` → `Zero` → `Initial`
2. Loop: `Initial` → `Random` → `One` → `Initial`
3. Loop: `Initial` → `Random` → `Zero` → `One` → `Initial`
4. Loop: `Initial` → `Random` → `One` → `Zero` → `Initial`
5. Loop: `Zero` → `One` → `Zero`

The first four loops are all safe from Zeno runs due to the local SNZ witness on the edge `Initial` → `Random`. A closer look at the fifth loop shows that this loop can not iterate because the two guards need conflicting valuations of `x` (`x == 0`, `x == 1`) and `x` is a local variable and can thus not be modified from an external source. The system is therefore free from Zeno runs. However, without analysis of the data variables the fifth loop will be detected as a loop that exhibits Zeno runs.

Conflicting loops. Let lp be a loop in a network of timed automata $|A$. A loop is *conflicting* if there exist two edges e_1, e_2 with respective guards g_1, g_2 involving local variables, such that there is no variable valuation v such that v $\models g_1$ and the variable valuation v$'$ resulting from sequential execution of the updates on the path from e_1 to e_2 satisfies g_2.

$$\nexists \langle \bar{l}, \mathrm{v} \rangle \in S[\mathrm{v} \models g_1 \wedge u_{n-1}(u_{n-2}(\dots u_1(\mathrm{v})\dots)) \models g_2]$$

A *conflicting loop* may safely be removed from the analysis result set as such a loop can never iterate during system execution. By definition, there is no variable valuation that satisfies the first guard and, after updates have been applied, also satisfies the second guard of the loop. Accordingly, the execution of the loop is impossible at run-time and thus can not contribute to Zeno runs.

4. ZenoTool

This chapter focuses on the implementation of the analyses given in chapter 3. The chapter is organized as follows: Section 4.1 deals with the usage of the implemented tool. It covers available options, limitations of the implementation and possible workarounds. Section 4.2 presents implementation details of the tool, open-source libraries used, and underlying algorithms. Section 4.3 deals with the software validation process by regression testing and the application of the run-time analysis tool Valgrind.[1] At last, section 4.4 shows the results we obtained by analyzing several real world case study models with ZenoTool carving out its strengths and weaknesses.

4.1. Usage

We implemented the analyses shown in chapter 3 in a stand-alone software called ZenoTool. Apart from the use of a few libraries (see section 4.2.3) ZenoTool was created from scratch and is written in the C++ programming language. C++ was our language of choice as it is generally known to perform well and as the analysis might be time-critical for complex models. ZenoTool is a command-line program reading UPPAAL model specification files. Typically, one creates a model using the editor provided by the UPPAAL graphical user interface. Note that ZenoTool only reads the more recently implemented XML model files introduced in UPPAAL version 3.4. ZenoTool is called in the following way:

```
ZenoTool [parameter, parameter, ...]  <modelfile>
```

Available parameters are given in the following.

4.1.1. Command-line Parameters

ZenoTool provides several command-line options to specify the exact analysis to perform. By default, the tool uses the synchronization matrix method (see section 3.2.2) to propagate the loop safety property. Elementary loops are detected by using a modified version of Tarjan's cycle detection algorithm [32] and no data variable heuristics are applied. The following command-line options are available:

- `-h` or `--help`
  Displays available command-line parameters and general usage information for ZenoTool.

[1] `http://valgrind.org/`

- `-v` or `--version`
  Displays the version of ZenoTool.

- `-s` or `--simple`
  Changes the output format of the analysis. By default, loops are printed including all the annotations on the edges. In complex models, loop annotations may be confusing and thus the output may be difficult to read. This command-line option prevents edge annotations from being printed.

- `-t` or `--tiernan`
  Changes the cycle detection algorithm to use the one proposed by Tiernan [33]. The algorithm does not handle multiple edges between the same two locations correctly and thus does not find all loops in such models. Also, generally the algorithm runs slower. Usage of this deprecated cycle detection algorithm is not recommended for those reasons.

- `-g` or `--group`
  Changes the loop safety propagation method to use Gómez' synchronization group approach (see section 3.2.1). Generally the synchronization group method will yield less accurate results at a faster processing speed when compared to the synchronization matrix method. If the analysis' run time is unsatisfactory switching to Gómez' method might improve performance.

- `-d` or `--dependent`
  Enables the data variable heuristic that detects a subset of safely dependent loops (see section 3.3.1). The analyzed model should be free from side effects to yield correct results. Therefore, neither user-defined functions nor assignments with side effects should be present in the model.

- `-c` or `--conflict`
  Enables the data variable heuristic that detects a subset of conflicting loops (see section 3.3.2). The analyzed model should be free from side effects to yield correct results. Therefore, neither user-defined functions nor assignments with side effects should be present in the model.

- `-r` or `--regression`
  Enables the final run of regression tests. Note that this option is mainly used for development and only makes sense in conjunction with the regression test model.

Note that multiple, enabled data variable heuristics will be applied in the order specified at the command line and a loop that qualifies for removal due to multiple heuristics will only be marked by the first matching heuristic. Enabling multiple heuristics at the same time though can create synergies such that the first heuristic removes a loop and because of this a second heuristic can be applied to remove even more loops. ZenoTool takes care of such synergies automatically by repeatedly executing the heuristics until no further improvements are possible.

4.1.2. Limitations and Workarounds

As presented in section 4.1 ZenoTool only reads UPPAAL XML model files introduced in UPPAAL version 3.4. If one happens to have an old UPPAAL model file (`ta`-file) a conversion needs to be done. The conversion process can be done by using the UPPAAL GUI. However, opening the old file and then saving it in the XML format will not solve the problem completely as the language specification also changed to accommodate minor inconsistencies in it. One notable change is that formerly multiple guards on an edge were separated by commas. However, the new format requires a conjunction operator instead. During the conversion process one needs to correct such formal errors. Hitting the key `F7` will help the user and mark outdated language grammar in the specification. Only models compliant to the language specification of UPPAAL version 4.0 in XML format are guaranteed to be understood by ZenoTool correctly.

ZenoTool does not yet understand all language features provided by UPPAAL and thus special attention needs to be given to the designated models. In particular one needs to refrain from using the following language constructs without taking a closer look at the results as certain constellations can lead to unsound analysis results and thus invalidate the conservative nature[2] of ZenoTool:

- **User-defined functions**
  ZenoTool ignores user-defined functions and therefore can not infer loop safety properties if guards or assignments are dependent on function calls and thus are not explicit. If a model uses user-defined functions the tool may fail to find all safe loops and yield a less accurate result set. The usage of data variable heuristics may even yield wrong results because possible variable valuations can not be inferred and thus loops may be removed due to seemingly missing variable valuations. Therefore, if the designated model uses user-defined functions and heuristics are enabled the analysis results need to be evaluated carefully as the analysis is then unsound.

- **Data records**
  Currently, ZenoTool does not parse data records. To ZenoTool, the usage of UPPAAL's `struct` construct renders unavailable all data contained in the so defined data record. References to such concealed data can not be evaluated by ZenoTool and the result of the analysis is influenced in the same way as user-defined functions influence it (see above).

- **Scalar variables**
  Data variable heuristics do not support scalar variables and their usage may thus decrease the accuracy of the heuristics because a loop that can be removed because of the scalar variable may not be identified correctly. The analysis remains sound though even when scalar variables are used.

[2] ZenoTool overapproximates the set of loops prone to Zeno runs.

- **Complex expressions**
  Expressions in UPPAAL's specification language are recursively defined and can get complex very quickly. ZenoTool currently only supports a very limited subset of the cases available. Expressions of the forms `+e`, `-e`, `e[e]`, `e + e`, and `e - e` can be correctly evaluated. Here, `e` denotes either an expression of the previously given forms, a reference to a variable, a natural number, or one of the keywords `true` or `false`. Note that assignments and boolean expressions are handled separately and are therefore not included in this list. Notable forms of expressions that are not understood and should be avoided are `e++`, `e--`, `++e`, `--e`, binary operators that are neither `+` nor `-`, and the case distinguishing construct `e?e:e`.

- **Selection statements**
  Selection statements generally work correctly as long as the variable name does not shadow any other template or global variable name. In such a case, the value of the template or global variable is used instead of the selection variable value. This behavior is not consistent with UPPAAL's where the selection statement variable hides other variables. The difference in shadowing treatment may render the selection statement without influence and thus compromise the analysis result. Ensuring uniqueness of selection statement variables prevents this problem and keeps the analysis sound.

All other language features are supported by ZenoTool. This includes bounded integers, booleans, clocks, channels, (multi-dimensional) arrays, custom type definitions, edge annotations, template parameters, and (automatic) template instantiation according to the system declaration.

When usage of ZenoTool is intended the user can assist the tool by providing a model that plays to ZenoTool's strengths. Such a model has the following properties, which were mostly derived from the list of unsupported features above:

- Local clock variables are declared local to templates and not globally.
- User-defined functions are not used.
- UPPAAL's struct construct is not used.
- Expressions are free from side effects (e.g., no use of `++`).
- Selection statement variables have unique names.
- Expressions do not contain on-the-fly case inspection using the `:?` operator.

A model complying with these properties can effectively be analyzed by ZenoTool including the currently implemented data variable heuristics.

4.2. Implementation Details

The implementation of the analysis is straight forward. At first the program configures its operation environment. A special class processes the command-line input and, depending on the specified options, classes implementing the according algorithms are instantiated. Next, the provided UPPAAL XML model file is parsed: We use TinyXML[3], a lightweight, stand-alone XML parsing library (see section 4.2.3) to create the XML data tree, which in turn is used to instantiate classes representing the timed automata network. The remaining system consists of a parsing subsystem and the analysis objects. The parsing subsystem is used during the automaton construction process to parse the variable declarations and the system declaration.[4] Individual classes are instantiated for every declared variable and correct (automatic) template parameter binding is achieved. Also, during the analysis step it is used to parse edge annotations.

After the initial phase finished the setup of the environment with the network of timed automata and all the involved variables, the actual model analysis is performed. At first, we extract all loops from the automata network. Then we determine the strong non-Zenoness and the safety properties for the loops. Afterwards we perform a safety propagation algorithm and at last we optionally apply data variable heuristics.

Figure 4.1 depicts the complete workflow of ZenoTool and its interaction with the parsing subsystem. As one can see after the initial selection of the different algorithms the tool works straight to the point calculating its analysis step by step while delegating parsing tasks to the parsing subsystem.

4.2.1. The Parsing Subsystem

The parsing subsystem features several classes that can be combined to form complex parsing structures. Basic string terminal parsing is achieved by instantiating one of the base parsers:

1. `CharParser`
   consumes a single specified character. (regular expression: `c`)

2. `RangeParser`
   consumes a single character that is between two specified characters inclusively. (regular expression: `[c1-c2]`)

3. `StringParser`
   consumes a single specified string. (regular expression: `string`)

4. `WhiteSpaceParser`
   consumes any amount of white spaces. (regular expression: `\s*`)

[3] `http://www.grinninglizard.com/tinyxml/`
[4] The system declaration specifies how to instantiate automata templates to form a network.

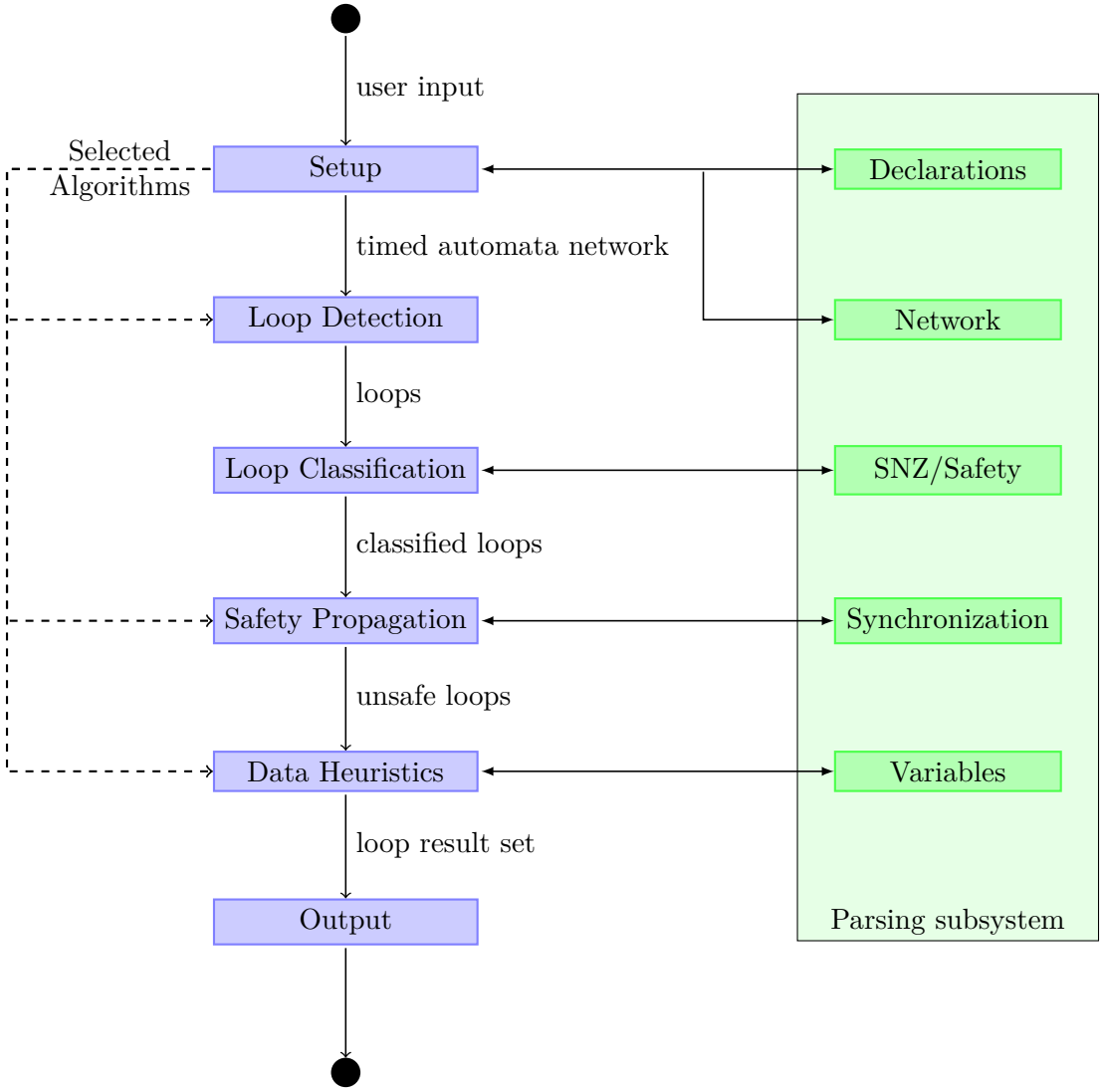


Figure 4.1.: ZenoTool's workflow

5. `AnyExceptParser`
   consumes characters until a specified character is reached. (regular expression: `[^c]*`)

6. `UntilParser`
   consumes characters until a specified character is consumed. (regular expression: `[^c]*c`)

All base parsers process the input string from left to right and consume characters depending on their options. The remaining string is available for further processing by a following parser object. Powerful parsers can be created by using the available combination parsers:

1. `ChoiceParser`
   The parser consists of any number of parsers and matches if any of the contained parsers matches. (regular expression: `p1|p2`)

2. `ConcatParser`
   The parser consists of any number of parsers and matches if all of the contained parsers match the input string in the specified order. (regular expression: `p1p2`)

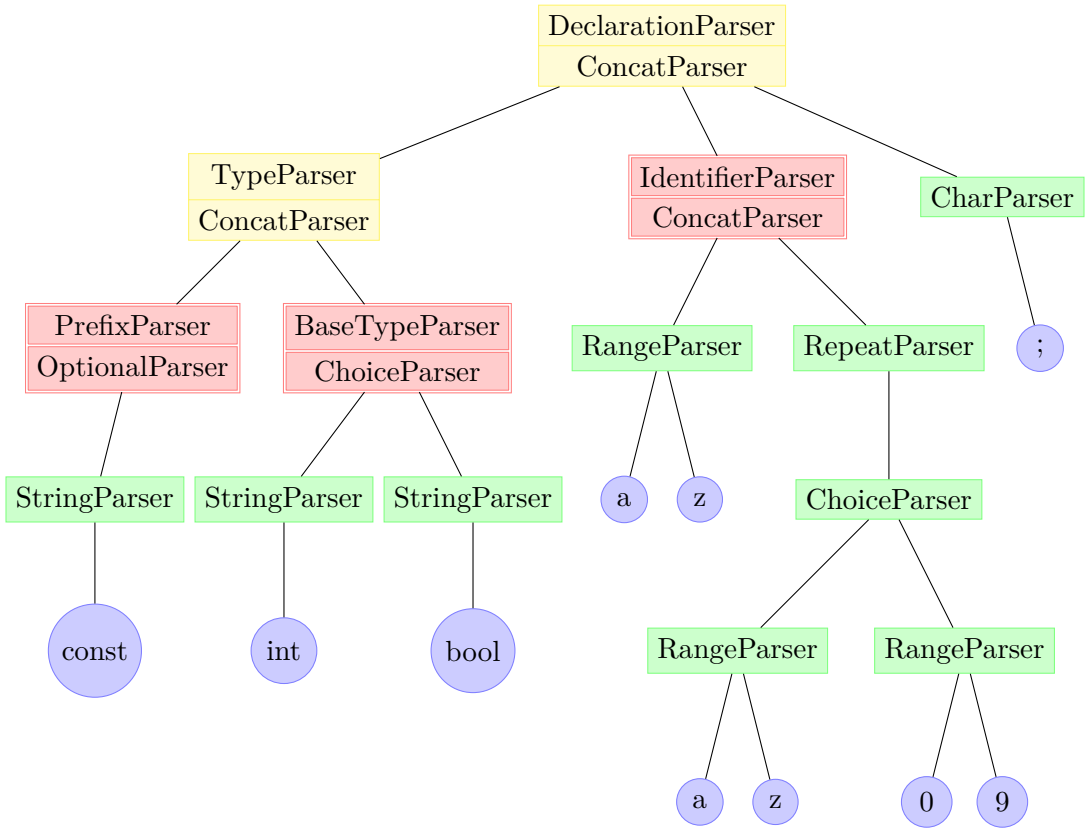
3. `OptionalParser`
   The parser wraps another parser that thereby becomes optional. (regular expression: `p?`)

4. `RepeatParser`
   The parser wraps another parser that thereby becomes optional and may be repeated. (regular expression: `p*`)

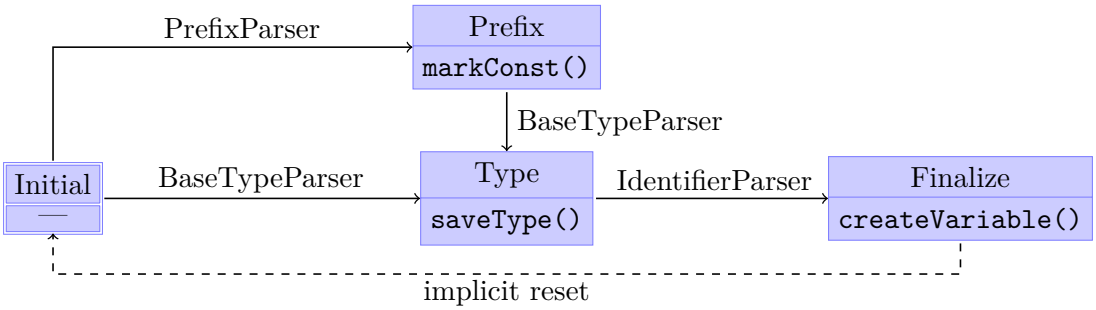
When defining a complex parser by combination, one can specify that the resulting parser should be a *reporting* one. Reporting parsers are necessary for the semantic analysis as just the syntactic analysis of the input string is insufficient. Therefore the user may specify a `ParserListener` object that will supervise the parsing process. During the parsing all reporting parsers that match a part of the input string will report to the listener class for it to handle the semantics. The correct listener class varies by the expectations on the input string type. For example, declaration strings need to be treated differently from edge annotations. Therefore the `ParserListener` class provides the user with an easy way to specify a state machine for the semantic analysis. Depending on the reports of parsers to the listener, the state machine changes states and executes necessary actions. Execution is achieved by enabling the user to link certain states to action handler routines that will execute when the state is reached.

Figure 4.2 shows a simplified example application of the parsing subsystem. In figure 4.2a the hierarchy of a combined parser is given.[5] The final `DeclarationParser` can be used to parse simple variable declarations of the form given by the regular expression

[5]Input to `ConcatParsers` is ordered from left to right.



(a) Declaration parser hierarchy



(b) Declaration semantics state machine

Figure 4.2.: Parsing system example

`(const)? (int|bool) [a-z][a-z0-9]*;`. Note that for brevity the handling of white spaces was omitted. In the image, custom-made parser objects are shown in yellow and red. Red parsers denote reporting parsers for the semantic analysis. Parsers marked green are base parsers provided by the system and blue nodes show the parameters for such base parsers. The `DeclarationParser` thus consists of three reporting parsers: one to handle the optional `const` prefix, one for the variable type (either `int` or `bool`), and one for the variable name that may start with any lower case letter and continue with lower case letters or numbers. The semantic analysis of such simple variable declarations could be achieved by a listener object implementing a state machine similar to the one given in figure 4.2b. In the initial state the listener awaits a report of either the `PrefixParser` or the `BaseTypeParser`. In the `PrefixParser` case an action handler is fired that saves that the analysis currently deals with a constant value. It then waits for the necessary `BaseTypeParser` report to obtain the variable type. In the `Type` state again an action handler is fired to save the type of the declared variable as still crucial information for the final variable instantiation is missing. The final report of the `IdentifierParser` provides the name of the variable and therefore the variable can then be instantiated using the information saved previously. Afterwards the state machine returns to its initial state because no further transitions are available and thus the parsing process is finished.

4.2.2. Algorithms

The algorithms involved during the analysis are presented in the following. Subsection 4.2.2.1 presents the algorithms used to find elementary loops in the timed automata network. Subsection 4.2.2.2 shows the implemented algorithms for strong non-Zenoness and safety property determination. Subsection 4.2.2.3 displays the safety propagation algorithms involving synchronization groups and matrices. And at last, subsection 4.2.2.4 goes into detail on the algorithms for the data variable heuristics.

4.2.2.1. Cycle Detection

ZenoTool features two different cycle detection algorithms. Both are quite similar and share the same general approach. They iterate over the locations of a directed graph in a depth-first search and use a stack data structure[6] to generate the elementary loops of the graph. To prevent cycles from being reported multiple times both algorithms initially establish an ordering of the locations dismissing duplicates by only allowing cycles whose locations are strictly ordered. The main difference between the algorithms is the way they determine which continuations of a path should be explored: Tiernan's method saves which continuations have already been taken for each location. Every time a path is extended a check occurs to prevent exploring the same continuation multiple times. Tarjan's method uses a more sophisticated approach. The algorithm removes locations from the graph under investigation as soon as a location can no longer contribute to the construction of additional elementary loops. Consequently, Tarjan's method generally

[6] A stack is a data buffer that organizes access in a last-in, first-out manner (LIFO).

is of lower computational complexity. For exact definitions of the algorithms we refer to the corresponding papers [32, 33].

Without modifications both algorithms return their results as sequences of locations in the directed graph. For our purposes this is insufficient because of the possible presence of multiple edges between the same two locations. Thus, our implementation of Tarjan's algorithm is an enhanced one: As the algorithm extends its paths by iteration on the outgoing edges of a location, all edges, also multiple ones, are explored by the algorithm already. Therefore, if multiple edges are present in the graph the result set already contains the correct amount of loops and we only inspect every location sequence and find matching edges between the locations. We then mark used edges and thus, on the next report of a similar loop, the subsequent edge will be used instead, ultimately covering all edges.

If at any time in the future the run time of the loop enumeration part of ZenoTool needs to be improved an implementation of the cycle detection algorithm by Szwarcfiter and Lauer [31] could accomplish such improvement as it usually performs even better than Tarjan's algorithm.

4.2.2.2. Strong Non-Zenoness and Loop Safety

ZenoTool's algorithm to determine the strong non-Zenoness property of a loop uses the previously discussed parsing subsystem. For every loop in the network we iterate over all edges of the loop and parse the guard and the update annotations that may be present on such edges by using a parser constructed for guards and updates. Both parsers report to a special listener for the SNZ property. The state machine of the listener is straightforward: For guards the detected sequence is an expression followed by a comparison string and another expression. For updates we only expect two expressions (with an assignment operator in between that is not reported). Upon detection of such a sequence an action handler is called for guards or updates respectively. The update handler first checks if the left-hand side is a clock variable (in contrast to a data variable). Then, the assignment value is resolved and added to the clock variable including the position of the edge. The guard handler at first checks if the guard constrains a clock; if the guard imposes a lower bound it saves the resolved lower bound for the clock variable. Again, the position of the edge in the loop is saved along with the value. After the parsing of all the edges of a loop has finished the clock variables the loop uses are retrieved from the listener. Using those we calculate if any of the clocks is an SNZ witness: We iterate over all lower bounds a clock possesses, retrieve the bound and the update that precedes it, and check if the update value is smaller than the bound. In such a case the clock variable is an SNZ witness for the loop and therefore the loop is strongly non-Zeno.

In a next step we determine if the strongly non-Zeno loops are also safe. To accomplish that we again iterate over the set of loops of the network. This time we skip loops that are not strongly non-Zeno, though. The main idea is to invalidate all SNZ witnesses of the loop. We therefore retrieve the set of witness clocks of the loop and iterate over the witnesses. If a clock is a local clock then the loop is safe and we can skip the clock.

Otherwise, we need to find an external update to the clock. By iteration on the loop set (and skipping the loop we are processing) we check every loop for an update to our witness. If we find an external update the SNZ witness clock is invalidated and thus we remove it from the witness set. After the iteration over the witnesses is finished, determination of the safety property is easy. If the SNZ witness set still contains any witness, the loop is not only SNZ but also safe.

Until now we only determined safety by locality or by the absence of external updates. However, loops are also safe, if all external updates originate from safe loops. Thus, an additional step is necessary. Because during this step the safety property should become transitive we use a fix point iteration approach. We again check all strongly non-Zeno loops that are unsafe. This time we consider the safety property of the loops the external updates originate from. A loop is marked safe if all external updates originate from safe loops. As long as a loop can be marked safe the fix point iteration continues to ultimately ensure transitivity of the safety property correctly.

4.2.2.3. Synchronization Methods

Both algorithms dealing with safety propagation, the synchronization group method and the synchronization matrix method, use the parsing subsystem to initially extract the synchronization data from the loops. Similarly to the SNZ property determination, the set of loops is iterated over and for every loop all edges' synchronization annotations are parsed. The responsible listener validates the synchronization channel and then assigns them to two groups for sending and receiving channels. After the parsing process both algorithms retrieve the two constructed channel sets and assign the sets to the corresponding loop. We thus establish bidirectional references for loops and channels: a mapping from loops to channels and from channels to loops is created indicating usage of the synchronization channels. These mappings are the base for both synchronization algorithms.

We now present our implementation of Gómez' synchronization group method [22]. Our implementation is divided in two parts and uses a fix point calculation approach. The first phase is an initialization step, in which three data structures are prepared. At first we create the initial synchronization group by adding all unsafe loops of the network to the group. Secondly, two mappings that map synchronization channels to the natural numbers, are created; one for sending channels and one for receiving channels. Those mappings count how often a channel is used by all the loops in the synchronization group for either sending or receiving actions. The mappings are later utilized to determine whether or not synchronization partners for a certain loop still exist within the synchronization group. The second phase then calculates the fix point: until no loop can be removed from the synchronization group anymore, loops are eliminated from the set. We check for loop removal by iterating over the channels a loop uses and querying the previously established mappings whether or not there is at least one matching synchronization partner in the synchronization group. If a loop happens to miss a partner the loop is removed from the synchronization group and the mappings are adjusted accord-

ingly by decreasing the synchronization count values of the respective channels. As soon as the fix point is reached, all loops' synchronization requirements are fulfilled and no loop can be removed anymore. Because we start with a maximal set in the beginning the analysis result is the maximal synchronization group regarding the amount of members.

For the newly proposed synchronization matrix method we refer to section 3.2.2 in the previous chapter. There, the construction of the loop representation vectors is explained in detail and it is shown how to obtain the (in)equation system. To solve the system we used the GNU Linear Programming Kit (GLPK)[7] (see section 4.2.3) and therefore the interesting algorithmic questions are already covered previously as we did not implement an equation solver by ourselves.

4.2.2.4. Data Variable Heuristics

Currently two data variable heuristics are implemented in ZenoTool: one that detects safely dependent loops (see section 3.3.1) and one for conflicting loops (see section 3.3.2). Both heuristics analyze whether or not guards of a loop can be satisfied using concrete data variable valuations. We therefore created a shared object that represents such guards and can be queried for satisfaction. For simplicity the object only considers guards that compare a data variable with a constant; an analysis comparing two variables is currently not featured and thus only a subset of safely dependent or conflicting loops is found. Alongside the guard representation object a matching parsing listener is provided that collects variable valuations and creates the representations. When analyzing update annotations the listener will try to resolve the data variable assignment and store the concrete values. If the concrete value can not be determined any value must be assumed, which generally results in a free data variable that can satisfy any guard. When analyzing a guard annotation all components (data variable reference, comparator string, and comparison constant) are checked for plausibility and accordingly a representation object is created. For such representation objects the listener can be asked if the currently contained variable valuation set satisfies the guard. Such queries are the main functionality of the guard representation object and listener classes. One should note that during the construction of the guard representations by the listener, guards are not simplified and only atomic guards are correctly parsed. For example, although $a < b$ and $\neg(a \geq b)$ are semantically equivalent the equivalence is currently not taken care of and the latter is not understood by ZenoTool.

We now present our algorithm that detects a subset of safely dependent loops. The elimination of such loops is achieved by a two-step algorithm. In the first phase the analysis parses all guard and update annotations for every loop in the main analysis result set.[8] The result is a set of guard representation objects for a certain loop and a listener that contains all possible data variable valuations of the loop. The set of guard representation objects is then reduced by querying the listener for variable valuations:

[7] `http://www.gnu.org/software/glpk/`

[8] Such a set only contains loops that may exhibit Zeno runs.

a guard is removed if there is no valuation for its constrained variable or if there is an assignment but it satisfies the guard. Such guards can not render loops safely dependent as we need to reset a variable to a value that does not satisfy the guard. After this initial phase we thus obtain a set of guard candidates for safely dependent loops. The actual property now depends on external updates to the variables. Therefore in the second phase we calculate unsafe updates to all data variables and remove safely dependent loops accordingly. A fix point iteration is utilized: we iterate over all loops in the unsafe loop set until no loop can be removed anymore. During an iteration we first parse all loops' update annotations. The parsing process yields a listener object containing all data variable valuations that are reachable by the execution of the unsafe loops. We then check all guard candidates for satisfaction. If a guard's variable requirements can not be fulfilled, the loop the guard object was constructed from can be removed and marked safely dependent. Variable valuations that satisfy the guard can only be obtained by execution of safe loops and thus the loop is safe at this point. Upon removal the fix point iteration starts over as the reachable variable valuations change when loops are removed.

The heuristic to detect conflicting loops is straight to the point. We check every loop in the main analysis result set exactly once: At first we parse all update and guard annotations of the loop to obtain the guard representation objects and the reachable variable valuations in the loop. If we constructed two or more guards we compare all pairs of guards. At first, it is checked whether or not two guards require a variable to have conflicting data variable valuations. Two guards conflict if they constrain the same variable and one of the following cases is present (we assume a to be the constrained variable and c_1 and c_2 the constraining constants of the guards):

- First guard: $c_1 < a$, Second guard: $a < c_2$, Constraint: $c_2 \leq c_1$
- First guard: $c_1 < a$, Second guard: $a \leq c_2$, Constraint: $c_2 \leq c_1$
- First guard: $c_1 < a$, Second guard: $a = c_2$, Constraint: $c_2 \leq c_1$
- First guard: $c_1 \leq a$, Second guard: $a < c_2$, Constraint: $c_2 \leq c_1$
- First guard: $c_1 \leq a$, Second guard: $a \leq c_2$, Constraint: $c_2 < c_1$
- First guard: $c_1 \leq a$, Second guard: $a = c_2$, Constraint: $c_2 < c_1$
- First guard: $a < c_1$, Second guard: $c_2 < a$, Constraint: $c_1 \leq c_2$
- First guard: $a < c_1$, Second guard: $c_2 \leq a$, Constraint: $c_1 \leq c_2$
- First guard: $a < c_1$, Second guard: $a = c_2$, Constraint: $c_1 \leq c_2$
- First guard: $a \leq c_1$, Second guard: $c_2 < a$, Constraint: $c_1 \leq c_2$
- First guard: $a \leq c_1$, Second guard: $c_2 \leq a$, Constraint: $c_1 < c_2$
- First guard: $a \leq c_1$, Second guard: $a = c_2$, Constraint: $c_1 < c_2$

- First guard: $a \neq c_1$, Second guard: $a = c_2$, Constraint: $c_1 = c_2$
- First guard: $a = c_1$, Second guard: $a = c_2$, Constraint: $c_1 \neq c_2$
- First guard: $a = c_1$, Second guard: $a < c_2$, Constraint: $c_2 \leq c_1$
- First guard: $a = c_1$, Second guard: $a \leq c_2$, Constraint: $c_2 < c_1$
- First guard: $a = c_1$, Second guard: $c_2 < a$, Constraint: $c_1 \leq c_2$
- First guard: $a = c_1$, Second guard: $c_2 \leq a$, Constraint: $c_1 < c_2$
- First guard: $a = c_1$, Second guard: $a \neq c_2$, Constraint: $c_1 = c_2$

If a conflicting pair of guards is found, we additionally check if there is a variable valuation such that the loop can satisfy any of the guards on its own. In such a case the conflicting property does not hold, because one guard can be satisfied externally and the second one by the guard itself possibly yielding a Zeno run. If the guards are not satisfiable internally the loop is removed and marked conflicting.

4.2.3. Libraries

ZenoTool is not completely created from scratch. It utilizes three different libraries; two are used directly and one indirectly. The used libraries are TinyXML[9], a lightweight library to read and write XML files, the GNU Linear Programming Kit (GLPK)[10] , a library for solving linear programs and related problems, and the GNU Multiple Precision Arithmetic Library (GMP)[11], a library for calculations with arbitrary precision. The latter is used by the GLPK for performance reasons. The program would work without it but then the GNU Linear Programming Kit would use its own, slower implementation of arbitrary precision calculations. In the next paragraphs we briefly explain each of the libraries.

TinyXML is an XML parsing library. It provides several classes to access various aspects of an XML file. Initially, parsing of an XML document constructs the XML Document Object Model (DOM). The DOM is a tree structure for the different parts in the XML document. Using the DOM it is possible to search certain elements for their attributes and contained data. We use TinyXML to parse UPPAAL's model specification files. The construction of the internal timed automata model is based on the extracted XML DOM.

The GNU Linear Programming Kit (GLPK) is an advanced library for solving linear programs and related problems. ZenoTool uses its features to solve the constrained equation system that is obtained when calculating propagation of the safety property by the synchronization matrix method. Specifically the mixed integer programming part (MIP) of the GLPK comes in handy in particular because it allows one to restrict certain

[9] `http://www.grinninglizard.com/tinyxml/`
[10] `http://www.gnu.org/software/glpk/`
[11] `http://gmplib.org/`

variables to integer values only. As the upper bound of loop iterations must be a natural number such restrictions are necessary for correct results. The GLPK uses advanced calculation methods such as *presolving* to relax the constrained equation system and sparse matrix representation methods to save memory. For usage information we refer to the excellent documentation provided with the library.

The GNU Multiple Precision Arithmetic Library (GMP) is used only indirectly by ZenoTool. The GLPK uses its features during the equation system solving process for performance reasons. The GMP provides its user with several features for computing with arbitrary precision. Not only are integers of arbitrary length supported but also rational numbers, which are represented using their nominator and denominator. Additionally, if the precision of the built-in *double* type of the C language is not high enough also floating point operations can be done with GMP.

4.3. Validation

During the development of ZenoTool two approaches were employed to validate its behavior. On the one hand we used regression testing to monitor the results of the analysis steps. On the other hand we used the open source tool Valgrind[12] to analyze ZenoTool for technical implementation faults like memory leaks.

The regression test suite covers all aspects of the main analysis and the two data variable heuristics presented. All aspects in this context refer to all special cases that may arise during the analysis steps including the invalidation of an SNZ property due to the update ordering and unresolved access of an array of synchronization channels. Also, the different language features of UPPAAL that ZenoTool supports are tested for their functionality. In total the test suite consists of 72 tests. 39 of them deal with the correct determination of strong non-Zenoness and safety using various language constructs. 18 test the synchronization propagation by evaluating the analysis result set and 15 evaluate the data variable heuristics results. The corresponding test model features 40 templates with a total of 165 loops to classify by ZenoTool. All loops' properties were determined by hand and the tests were generated accordingly such that the tests that validate ZenoTool yield the same correct results.

The different aspects of the software, which test cases are existent for, are listed in the following:

- **Loop properties**
  - Safe/SNZ loop classification
  - Unsafe/NSNZ loop classification
  - Safe/SNZ loop due to order of clock updates
  - Unsafe/NSNZ loop due to order of clock updates

[12] `http://valgrind.org/`

  - Safe/SNZ loop in spite of external clock update (safe update)
  - Unsafe/SNZ loop due to unsafe external clock update
  - Unsafe/NSNZ loop due to invalid guard (no lower bound)
  - SNZ/NSNZ loop classification due to multiple edges
  - Detection of multiple SNZ witnesses for an SNZ loop
- **Synchronization and propagation**
  - Sending on broadcast channels
  - Receiving on broadcast channels
  - Sending on binary channels
  - Receiving on binary channels
  - Transitivity of safety propagation
- **Data variable heuristics**
  - Variable valuation collection (concrete values, ambiguous values)
  - Guard evaluation using variable valuations
  - **Safely dependent loops**
    - Safely dependent loop using integers
    - Not safely dependent loop using integers
    - Safely dependent loop using booleans
    - Not safely dependent loop using booleans
    - Not safely dependent loop due to missing reset
    - Transitivity of the safely dependent property
  - **Conflicting loops**
    - Conflicting loops (all combinations of two guards)
    - Not conflicting loops (all combinations of two guards)
    - Not conflicting loop due to internal guard satisfaction
  - External heuristics influence (safely dependent loop due to conflicting one)
- **Language features**
  - Declaration and usage of constant values
  - Declaration and usage of custom types
  - Declaration and usage of arrays
  - Declaration and usage of multi-dimensional arrays
  - Passing of call-by-value template parameters

  - Passing of call-by-reference template parameters
  - Automatic binding of free template parameters

- **Parsing subsystem**
  - Parsing of declarations (variable instantiation and initialization)
  - Parsing of guards (different relational operators, conjunctions)
  - Parsing of updates (different assignment operators)
  - Parsing of synchronization labels (valid channels, send/receive)

The test cases not only include tests for each of the tool aspects individually but also tests that check multiple aspects. For example the test case classifying a loop with an unsafe external update also makes use of arrays testing the array resolving functionality as well. Overall ZenoTool fulfills all requirements imposed by the test suite and can therefore be considered to behave correctly in the bounds of the covered testing space. A few examples of test cases can be found in appendix A.

Implementation errors on the technical side were minimized by making ZenoTool subject to the run-time analysis of Valgrind. Valgrind's default analysis tool is *memcheck*, a tool that detects memory errors. Such errors include trying to access invalid memory e.g. buffer overruns on heap blocks, accessing memory that has already been freed, usage of uninitialized variables, incorrect memory freeing, and general memory leaks. We applied Valgrind 3.6.1 Debian's memcheck tool to ZenoTool using the `-leak-check=full` parameter to obtain a complete memory leak log. The analysis results show that no memory leaks occur in our implementation. Also, no invalid write or reads to variables are present indicating that neither uninitialized variables have influence on ZenoTool's analysis nor already disposed memory is used as input. Correct memory allocation and freeing is asserted and therefore a consistent program state is achieved. Also, intrusion attacks against the software like double-free attacks are thus not possible. Stack overflows could not be found either in ZenoTool. In total no technical mistakes detectable by Valgrind were present in ZenoTool and thus ZenoTool's implementation can be considered free from such errors.

4.4. Experiments

To determine the accuracy and efficiency of ZenoTool's analysis a wide variety of model specifications was analyzed. Models include simple example models delivered with UPPAAL, benchmark models showing UPPAALs performance, and advanced models of (communication) protocols that were developed by other scientific case studies. Our selection of models covers a broad spectrum of use cases and thus allows determining general qualities of ZenoTool.

To run the tests, we wrote two small driver programs. The test for accuracy runs ZenoTool with the following configurations:

- **Safety propagation method**: Synchronization Groups
  Data variable heuristics: No heuristic
- **Safety propagation method**: Synchronization Groups
  Data variable heuristics: Safely dependent loops
- **Safety propagation method**: Synchronization Groups
  Data variable heuristics: Conflicting loops
- **Safety propagation method**: Synchronization Groups
  Data variable heuristics: Both heuristics
- **Safety propagation method**: Synchronization Matrix
  Data variable heuristics: No heuristic
- **Safety propagation method**: Synchronization Matrix
  Data variable heuristics: Safely dependent loops
- **Safety propagation method**: Synchronization Matrix
  Data variable heuristics: Conflicting loops
- **Safety propagation method**: Synchronization Matrix
  Data variable heuristics: Both heuristics

ZenoTool's run time is determined by executing the analysis ten consecutive times and saving the fastest, the slowest, and the average run times. Valgrind's *massif* tool was used to find the peak heap memory consumption of ZenoTool.[13] All experiments were done using an Intel Core 2 Duo CPU running at 3.33GHz with 8GB of RAM on an Ubuntu 11.10 system.

Our test suite consists of 13 models. Three of them are example models distributed with UPPAAL, namely the *train-gate8* model, the *fischer6* model and the *bridge* problem. Four of them, the *csmacd2* and *csmacd32* models, the *fddi32* model, and the *bocdp* model, are used to benchmark UPPAAL's performance but also originate from scientific case studies. The remaining models were explicitly derived from scientific case studies. In total eight of the models are free from Zeno runs and the remaining five exhibit Zeno runs in one or multiple ways.

We now discuss the models in detail, including their Zeno behavior; the results of the analysis are summarized in table 4.1.

4.4.1. Models

Bridge model. The bridge model is a simple example model delivered with UPPAAL. It models four soldiers that want to pass an unlit, damaged bridge and move at different

[13] Note that additionally to the heap memory some static memory will be used, which is not accounted for here.

speeds. However, the soldiers only have a single torch that enables a maximum of two soldiers to cross the bridge safely at a time. The system has two templates, one for the torch and one for a soldier. The soldier template is parameterized to accommodate multiple walking speeds. The model has two loops in the torch model and one loop per soldier instance yielding a total of six loops. The soldier loops are strongly non-Zeno and the torch loops are not. However, due to synchronization all loops of the torch automaton also become safe; the model is free from Zeno runs. ZenoTool also finds this result for all configuration combinations.

Lip Synchronization model. The lip synchronization model was extracted from a case study by Bowman and Gómez [11]. Lip synchronization in this case refers to the process of merging and synchronizing an audio and a video stream, which were obtained separately. The transmission channel the data is received on has varying throughput and thus synchronization of the audio and video needs to be paired up correctly. The UPPAAL model for lip synchronization has a total of 12 templates and 13 loops. Six of the loops are not strongly non-Zeno, but become safe by synchronization. Therefore, the system is free from Zeno runs. The absence of Zeno runs is validated by ZenoTool for all configuration combinations.

Train Gate model. The train gate model is an example model also delivered with UPPAAL. It models the problem of multiple trains approaching and crossing a bridge that can only be used by one train at a time. Because trains can not stop immediately strict timing constrains apply. The system consists of two templates, one for the bridge and one for a train. In our case eight trains were instantiated. In contrast to the three loops in the bridge template all the two loops in the train template are strongly non-Zeno and safe. Nevertheless the system is free from Zeno runs due to synchronization. Because the bridge template involves a queue-like data structure modeled with user-defined functions ZenoTool's data heuristics were not applicable to the model, but even without them ZenoTool correctly classifies the model safe with both synchronization methods (synchronization groups, synchronization matrix).

Token Ring FDDI Protocol model. The token ring fiber distributed data interface protocol model is a benchmark model for UPPAAL and can be obtained from the corresponding website.[14] It models a token distribution protocol in a local area network composed of N symmetric stations organized in a ring. We analyzed a model with 32 stations. Two different templates need to be considered: the station template and the ring template. The station template consists of four safe loops and the ring template only has a single unsafe loop. Again, the model is free from Zeno runs because of safety propagation. ZenoTool yields the same result for all configuration combinations.

Bang & Olufsen Collision Detection Protocol model. The Bang & Olufsen collision detection protocol model was developed in a case study [23] but it is also used as a benchmark for UPPAAL.[15] The protocol exchanges control information between different audio/video units in the company's product line. The system models two units

[14] `http://www.uppaal.org/benchmarks/`
[15] `http://www.uppaal.org/benchmarks/`

represented by four templates and an additional template for the communication bus. Of the templates only one template per unit is made up from safe loops. All other loops are unsafe but become safe due to synchronization yielding a system that is free from Zeno Runs. All configuration combinations for ZenoTool confirm this result.

TDMA Protocol Start-Up Mechanism model. The TDMA protocol start-up mechanism model was derived from a case study [29]. It models the communication protocol of the Dependable Architecture for Control of Applications with Periodic Operation (DACAPO), which is intended for small distributed systems. For communication each node is assigned a time slot, a so called TDMA slot. The start-up mechanism, i.e., the synchronization of the time slots of all participants, needs to be carefully crafted and thus it is verified in the case study. The model consists of a template for the communication bus, a template for the communication nodes, and a test automaton for verification purposes. Due to strict timing constraints all loops in the station template are safe from Zeno runs, however, the test automaton has a single unsafe loop and the bus model has eight unsafe loops. All but two loops become safe by safety propagation. Closer examination of the pair of unsafe loops shows that they can not contribute to Zeno runs as one loop can only iterate in an erroneous state that is not reachable, which can be shown by UPPAAL. Altogether, the model is thus free from Zeno runs. ZenoTool for all configuration combinations, however, finds the pair of unsafe loops as it can not recognize one loop is in unreachable state space. Though, ZenoTool's result enables the user to verify the absence of Zeno runs fast by hand.

Gearbox model. The model of a gearbox with the respective engine and controllers was developed in a case study by Lindahl, Pettersson, and Yi [28]. Timing constraints of gear change requests and corresponding torques in the engine are modeled and verified. Five different templates form the system: a gear controller template, a gearbox template, a clutch template, an engine template, and an interface template for gear changes. Concerning safety, the clutch, the engine and the gearbox templates are free from Zeno runs. The gear controller template has four unsafe loops and the interface template is completely constructed from six unsafe loops. After considering safety propagation only one loop of the gear controller remains unsafe and all interface loops still are unsafe and can synchronize with the controller. Closer examination yields that Zeno runs can not occur: for the loops to iterate the gear controller template requires two data variables to be less than or equal to zero. The interface loops provide the variable valuations; but none that would allow the controller loop to iterate is present. Thus, the system is free from Zeno runs. ZenoTool's analysis finds the seven unsafe loops using any of the configurations. The data variable heuristics are not sufficient to eliminate these remaining loops. However, proving the absence of Zeno runs given the analysis result by hand is easy because of the simple synchronization scenarios involved and the small result set.

Carrier Sense, Multiple-Access with Collision Detection Protocol models. The carrier sense, multiple-access with collision detection protocol manages the assignment of a single communication bus to multiple competing stations. Collisions occur when multiple stations send at the same time. Correct collision handling is also specified in the CSMA/CD protocol. We analyzed two different UPPAAL models for the

protocol. The first one is an UPPAAL benchmark model[16] with a varying amount of stations. We analyzed the model for two and 32 stations respectively. The second model is derived from a case study by Bowman and Gómez [12] and was analyzed for two stations only.

We first focus on the case study model by Bowman and Gómez. The model has three templates, one for a station, one for a communication interface, and a last one for the communication medium. The station and the interface templates are instantiated twice, once for each station. Except for two loops in the station template all loops are unsafe. Synchronization propagation reduces the amount of unsafe loops to four, two per station. Investigation of the loops shows that the model indeed exhibits Zeno runs. ZenoTool correctly finds the four responsible loops with all configuration combinations.

The model for benchmarking purposes is constructed differently. The system consists of a single template for the communication bus and one template per station. Concerning safety, the bus template consists of three unsafe loops and the station templates have three safe and three unsafe loops. When considering 32 stations two of the bus template loops become safe by safety propagation. In contrast all station loops remain unsafe and indeed Zeno runs can occur due to them. The analysis result features 97 unsafe loops in the model, three per station plus a single one for the bus. ZenoTool finds those loops with both safety propagation methods. However, when considering only two stations, ZenoTool yields an improvement over the expected seven unsafe loops. In this case the synchronization matrix method only yields three loops, one per station and bus template. Closer examination shows that the reduction to two stations eliminated some synchronization possibilities rendering some loops unable to contribute to Zeno runs. The more accurate results set of three loops attained by the synchronization matrix is therefore correct and is an improvement over Gómez' synchronization group method.

Fischer Protocol model. The Fischer protocol is a very basic protocol to guarantee mutual exclusion. It was firstly described by Lamport [26]. The UPPAAL model is an example delivered with UPPAAL. It consists of a single template that models a participant. In our case six participants were instantiated. The template has two loops, a safe and an unsafe one. As the model uses no synchronization, unsafe loops remain unsafe. By manual analysis, however, it becomes obvious that the model does not exhibit Zeno runs: the unsafe loops can only iterate when they are paired with a safe loop. Accordingly ZenoTool's analysis proves the absence from Zeno runs if the safely dependent data variable heuristic is enabled. Otherwise the six unsafe loops, one per participant, are reported. For this model the data variable heuristic improves the accuracy of Gómez' original method and our synchronization matrix approach.

Biphase Mark Protocol model. The biphase mark protocol is commonly used for communication at the physical ISO/OSI layer to transmit reliably bit strings of arbitrary lengths. Our model was obtained from a case study by Vaandrager and de Groot [36] that determines clock tolerances in the biphase mark protocol of communicating partners. The model is made up of an encoder and a decoder template, two clock templates, a

[16] `http://www.uppaal.org/benchmarks/`

wire template, a sampler template, and a test automaton. Only the two clock templates and a single loop in the wire are directly safe from Zeno runs. As all but two loops directly depend on the clock templates, safety propagation eliminates all but those two unsafe loops from the result set. One can see that one of these loops can not advance as it is linked to a clock template using data variables. The other loop, however, indeed exhibits Zeno runs, though the Zeno run is intentional to model random bit flips on the wire. ZenoTool's results are as expected: when the safely dependent data variable heuristic is enabled the single intentional Zeno run loop is returned; otherwise the result set also contains the second, indirectly linked loop.

Zeroconf Protocol model. The zero configuration protocol specifies an algorithm for nodes to obtain valid IP addresses while ensuring mutual exclusion, i.e., no duplicate IP addresses in the system. In a case study by Gebremichael, Vaandrager, and Zhang [16] the protocol was analyzed using UPPAAL. We derived our model from their case study. The model features four different templates. Of the total of 22 loops, 8 are safe and by safety propagation no additional loop can be deemed safe. The analysis shows that the model is prone to Zeno runs. Considering ZenoTool, the model unfortunately makes use of UPPAAL's data record feature, which is currently not supported by ZenoTool. Also, user-defined functions are present in the model and therefore ZenoTool's data variable heuristics are not applicable to the model. Still the base analysis of ZenoTool succeeds to find the set of 14 unsafe loops that may contribute to Zeno runs.

A few examples of our analyzed case study models can be found in appendix B.

4.4.2. Accuracy

Our analysis succeeded to prove six of the eight models safe from Zeno runs. For the remaining two Zeno run free models the analysis' result set consisted of only a handful of false positives. Because of the small result sets it was easy to infer the absence of Zeno runs by hand, allowing our tool to classify all eight models correctly with little or no manual work at all. Concerning the five models that can exhibit Zeno runs for all but one model small sets of loops were correctly found to be responsible for the occurrence of Zeno runs. These sets could in future be used by application developers to refine the models to get rid of the Zeno runs or to help reason why the usage of Zeno runs is an acceptable approximation.

Comparing our enhanced analysis to Gómez' base method, our more powerful synchronization matrix method for safety propagation yielded an improvement in accuracy only for a single model. In these cases the conflicting loops data heuristic found not a single conflicting loop indicating well thought-out models. However, the safely dependent heuristic succeeded to eliminate in total eight loops in three different models yielding a more accurate result. In one case the heuristic managed to prove the absence of Zeno runs in the model, which Gómez' method was not able to.

An overview of the obtained accuracy results can be seen in table 4.1.

Model	Zeno Runs	Sync Groups				Sync Matrix			
bridge.xml	✓	0	0	0	0	0	0	0	0
lipsync.xml	✓	0	0	0	0	0	0	0	0
train-gate8.xml	✓	0	ϟ	ϟ	ϟ	0	ϟ	ϟ	ϟ
fddi32.xml	✓	0	0	0	0	0	0	0	0
bocdp.xml	✓	0	0	0	0	0	0	0	0
tdma.xml	✓	2	2	2	2	2	2	2	2
gearbox.xml	✓	7	7	7	7	7	7	7	7
csmacd.xml	✗	4	4	4	4	4	4	4	4
csmacd2.xml	✗	7	7	7	7	3	3	3	3
csmacd32.xml	✗	97	97	97	97	97	97	97	97
fischer6.xml	✓	6	0	6	0	6	0	6	0
bmp.xml	✗	2	1	2	1	2	1	2	1
zeroconf.xml	✗	14	ϟ	ϟ	ϟ	14	ϟ	ϟ	ϟ
Heuristics Used			**D**	**C**	**D C**		**D**	**C**	**D C**

The Zeno run column indicates whether or not the model is free from Zeno runs (✓: free, ✗: not free). The numbers indicate the amount of loops prone to Zeno runs ZenoTool found. The ϟ symbol indicates a certain configuration combination was not applicable for a specific model. The character **D** displays usage of the safely dependent data variable heuristic; the character **C** stands for the conflicting loop heuristic. The Sync Matrix columns show the results of our enhanced detection approach while the Sync Groups columns show results obtained by Gómez' base method.

Table 4.1.: Analysis accuracy results obtained by ZenoTool

4.4.3. Performance

Concerning ZenoTool's performance for our performance checks we compared Gómez' synchronization group approach without heuristics to our synchronization matrix method using both data heuristics. We decided on this setup to emphasize that even if the work load is additionally increased because of the data heuristics, ZenoTool's performance scales well. Our performance results can be seen in table 4.2.

Generally all analyses run in less than a second even for complex models. Thus, time is not a major factor when using ZenoTool. Memory consumption generally also is quite low. During our experiments the heap memory consumption barely exceeded two megabytes, which is a very low value.

When taking a closer look at the performance results, comparing Gómez' method to ours the run times differences are nearly negligible. Only the *csmacd32* model shows a significant difference in run time. Our synchronization matrix approach including the

heuristics takes nearly twice the time than the original method. However, this is an expected result because the model benefits from the synchronization matrix approach. In the model using only two participants the analysis accuracy is improved due to the synchronization matrix calculation. This improvement shows that the constructed equation system is not trivial. Accordingly in the model using 32 participants the equation system to solve is significantly bigger and still not trivial. Thus, the solving process is more complex and takes more time.

	Sync groups, no heuristics				**Sync matrix, both heuristics**			
Model	**min**	**max**	**avg**	**memory**	**min**	**max**	**avg**	**memory**
bridge.xml	6	10	9	587	10	11	10	671
lipsync.xml	11	11	11	792	11	13	12	881
train-gate8.xml	9	15	12	620	ϟ	ϟ	ϟ	ϟ
fddi32.xml	77	86	80	1911	79	85	81	2011
bocdp.xml	172	183	175	1140	176	189	179	1221
tdma.xml	87	88	88	866	87	98	89	951
gearbox.xml	19	21	20	872	25	28	26	955
csmacd.xml	10	12	11	629	8	13	10	723
csmacd2.xml	11	11	11	633	9	15	10	721
csmacd32.xml	48	51	49	1881	95	98	96	1961
fischer6.xml	11	12	11	562	11	17	15	641
bmp.xml	11	15	12	684	13	14	13	764
zeroconf.xml	14	22	18	704	ϟ	ϟ	ϟ	ϟ
Units	[msecs]			[kb]	[msecs]			[kb]

All run times are in milliseconds and rounded up. Memory consumption is measured in kilobytes and is also rounded up. The given memory consumption is the peak heap memory consumption. ZenoTool additionally needs a certain amount of static memory that is not accounted for here. The ϟ symbol again indicates that a certain configuration combination was not applicable for a model.

Table 4.2.: Analysis performance of ZenoTool

Generally, we expected the performance to take a more significant hit due to the nature of the equation solving process involved in the synchronization matrix method. However, the results of the *csmacd32* model indicate that most of our test models generate equation systems that are very easy to solve for GLPK and thus run time does not increase as much. Models that benefit from the synchronization matrix accuracy improvement probably will perform worse than our results show.

The performance of the data variable heuristics depends on the accuracy of the previous analysis result set. As most experiment models returned small sets of unsafe loops the heuristics seem to have a small effect on ZenoTool's run time. Models that have

many unsafe loops without considering data variables probably will also see a drop in performance compared to our performance results.

Altogether ZenoTool's performance, however, is still very good as it provides its results nearly instantly to the user. When ensuring time divergence of a model using our static analysis results can be obtained at a much faster rate compared to using a test automaton in UPPAAL.

4.4.4. Conclusion

Considering all the facts gathered so far, ZenoTool's static analysis provides accurate, yet sometimes overapproximated results while still using few resources for its execution. The synchronization matrix approach always has at least the same analysis accuracy as Gómez initial Zeno run detection method. In special cases even an improvement in accuracy may be attained. From our performance experiments one can see that even though we use more machinery in our enhanced approach the synchronization matrix method performs well and compared to using synchronization groups for safety propagation the hit in performance is small.

Models that benefit from our enhanced approach the most have loops that make extensive use of synchronization. Loops synchronizing multiple times even on the same channel reduce possible synchronization scenarios and the synchronization matrix method detects such scenarios accurately.

Regarding the safely dependent loop heuristic models that use guards with data variables all have the possibility to greatly benefit from its usage. The extension of safety propagation by using linked data variables provides means to effectively improve Zeno run detection. A more general data flow analysis could further improve this result.

In contrast the conflicting loops heuristic should be seen as a sanity check because generally the presence of conflicting loops in models is not desirable as such a model becomes confusing and unclear most of the time. While it may improve accuracy of Zeno run detection in some models restructuring of such a model might be the better solution.

5. Conclusions and Future Work

In this thesis we discussed the influence of Zeno runs on UPPAAL specifications and presented detection methods to check for their absence in such models. We enhanced Gómez' initial Zeno run detection method that calculates synchronization groups to deal with safety propagation by introduction of a newly developed synchronization matrix. In contrast to the synchronization group method, which only checks for available partners, the synchronization matrix approach eliminates impossible synchronization scenarios by also considering the amount of necessary synchronization partners and thus improves safety propagation. In addition, we developed two data variable heuristics to improve the gained analysis accuracy even further. The safely dependent loop heuristic extends the concept of safety propagation to data variables assuring that variable valuations necessary for Zeno run iterations can only be attained by iterating over safe loops. Respectively, the conflicting loop heuristics achieves the removal of loops that are unable to iterate due to guards that require conflicting data variable valuations. ZenoTool, a command-line tool implementing these analysis methods, was developed and applied to several case studies to empirically evaluate the improvements of our enhanced detection approach.

The experiment results indeed validate our expectations. Due to the static nature of the analyses used, ZenoTool provides sound analysis results in a timely manner and in some cases our enhancements to Gómez' Zeno run detection method yielded improvements in the analysis accuracy. Loops that formerly were falsely considered to exhibit Zeno runs were shown to be safe and therefore absent from the analysis result set of our modified approach. In this context, the synchronization matrix method is most useful if the loops in the analyzed model require a multitude of synchronizations; specifically models with repeated use of the same synchronization channel benefit most. Also, performance apparently is no issue as ZenoTool's run time is generally within a few seconds and scales well regarding its memory consumption. Concluding, our static analysis provides sound, easy-to-use Zeno run detection functionality for a broad variety of UPPAAL specification models. Thus, it contributes to gaining confidence in the correctness of specifications, enhancing UPPAAL's explanatory power thereby.

However, ZenoTool is still work in progress and in the future, several improvements could be made to it. The most obvious one is establishing feature completeness regarding UPPAAL's specification language as currently ZenoTool is still missing support for certain constructs like data records or complex expressions.

A more complicated task concerning ZenoTool would be the incorporation of user defined functions, though this would probably require the introduction of a complete data flow analysis into the tool. Such a fully-fledged data flow analysis could also be incorpo-

rated into the presently implemented data variable heuristics significantly augmenting their power and the confidence in them. Thus, extending ZenoTool with a data flow analysis module is beneficial and should be considered in the future.

Another development possibility would be to create an interface from ZenoTool to UPPAAL. An additional analysis module in ZenoTool could make use of such an interface and check detected Zeno runs for their reachability as Zeno runs exhibiting loops may be in unreachable state space of the model and thus have no influence on the behavior of the model. Such a module could discard falsely detected Zeno loops depending on their reachability in the model state space further increasing the accuracy of the analysis result. Also, as Zeno runs are not necessarily harmful to a model the interface could be used to classify the Zeno runs regarding to their possibility to create timelocks.

Bibliography

[1] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking for real-time systems. In *Proceedings of the Fifth IEEE Logic in Computer Science (LICS), 1990*, pages 414 –425, jun 1990.

[2] Rajeev Alur and David Dill. Automata for modeling real-time systems. In Michael Paterson, editor, *Automata, Languages and Programming*, volume 443 of *Lecture Notes in Computer Science*, pages 322–335. Springer Berlin / Heidelberg, 1990. 10.1007/BFb0032042.

[3] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[4] Gerd Behrmann, Patricia Bouyer, Kim G. Larsen, and Radek Pelnek. Lower and upper bounds in zone based abstractions of timed automata, 2004.

[5] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. In M. Bernardo and F. Corradini, editors, *International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004. Revised Lectures*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–237. Springer Verlag, 2004.

[6] Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, John Håkansson, Paul Pettersson, Wang Yi, and Martijn Hendriks. UPPAAL 4.0. In *Quantitative Evaluation of Systems (QEST)*, pages 125–126, 2006.

[7] Sébastien Bornot, Joseph Sifakis, and Stavros Tripakis. Modeling urgency in timed systems. In *International Symposium: Compositionality - The Significant Difference*, pages 103–129. Springer, 1997.

[8] Howard Bowman. Modelling timeouts without timelocks. In *ARTS'99, 5th International AMAST Workshop on Real-time and Probabilistic Systems*, volume 1601 of *Lecture Notes in Computer Science*, pages 334–354. Springer-Verlag, May 1999.

[9] Howard Bowman. On time and action lock free description of timed systems. Technical Report 16-99, Computing Laboratory, University of Kent at Canterbury, December 1999.

[10] Howard Bowman. Time and action lock freedom properties of timed automata. In M. Kim, B. Chin, S. Kang, and D. Lee, editors, *Formal Techniques for Networked and Distributed Systems*, pages 119–134. Kluwer Academic Publishers, August 2001.

[11] Howard Bowman, Giorgio Faconti, Joost-Pieter Katoen, Diego Latella, and M. Massink. Automatic verification of a lip-synchronisation protocol using UPPAAL. *Formal Aspects of Computing*, 10:550–575, 1998.

[12] Howard Bowman and Rodolfo Gómez. How to stop time stopping. *Formal Aspects of Computing*, 18:459–493, 2006. 10.1007/s00165-006-0010-7.

[13] Conrado Daws, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. The tool KRONOS. In *Proceedings of the DIMACS/SYCON workshop on Hybrid systems III : verification and control: verification and control*, pages 208–219, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc.

[14] E. Allen Emerson and Edmund M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241 – 266, 1982.

[15] Biniam Gebremichael and Frits Vaandrager. Specifying urgency in timed I/O automata. In *Proceedings of the third IEEE International Conference on Software Engineering and Formal Methods (SEFM 2005)*, pages 64 – 73, sept. 2005.

[16] Biniam Gebremichael, Frits Vaandrager, and Miaomiao Zhang. Analysis of the zeroconf protocol using UPPAAL. In *Proceedings of the 6th ACM & IEEE International Conference on Embedded Software*, EMSOFT '06, pages 242–251, New York, NY, USA, 2006. ACM.

[17] Rodolfo Gómez. *Verification of Real-Time Systems: Improving Tool Support*. PhD thesis, Computing Laboratory, University of Kent, October 2006.

[18] Rodolfo Gómez. Verification of Timed Automata with Deadlines in UPPAAL. Technical Report 2-08, Computing Laboratory, University of Kent, June 2008.

[19] Rodolfo Gómez. A compositional translation of timed automata with deadlines to UPPAAL timed automata. In *Proceedings of the 7th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS 2009)*, FORMATS '09, pages 179–194, Berlin, Heidelberg, 2009. Springer-Verlag.

[20] Rodolfo Gómez and Howard Bowman. Discrete Timed Automata and MONA: Description, Specification and Verification of a Multimedia Stream. In H Konig, M Heiner, and A Wolisz, editors, *Formal Techniques for Networked and Distributed Systems - FORTE 2003. Proceedings of the 23rd IFIP WG 6.1 International Conference*, number 2767 in LNCS, pages 177–192, Berlin, Germany, September 2003. Springer.

[21] Rodolfo Gómez and Howard Bowman. Discrete Timed Automata. Technical Report 3-05, University of Kent, Computing Laboratory, February 2005.

[22] Rodolfo Gómez and Howard Bowman. Efficient Detection of Zeno Runs in Timed Automata. In J.-F. Raskin and P.S. Thiagarajan, editors, *Proceedings of the 5th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS 2007)*, volume 4763 of *LNCS*, pages 195–210, Salzburg, Austria, October 2007. Springer.

[23] Klaus Havelund, Arne Skou, Kim G. Larsen, and Kristian Lund. Formal modeling and analysis of an audio/video protocol: an industrial case study using UPPAAL. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, RTSS '97, pages 2–13, Washington, DC, USA, 1997. IEEE Computer Society.

[24] Martijn Hendriks, Gerd Behrmann, Kim G. Larsen, Peter Niebert, and Frits Vaandrager. Adding symmetry reduction to UPPAAL. In *Proceedings of the First International Workshop on Formal Modeling and Analysis of Timed Systems (FORMATS 2003)*, volume 2791 of *LNCS*, pages 46–49. Springer-Verlag, 2004.

[25] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111:394–406, 1992.

[26] Leslie Lamport. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 5(1):1–11, January 1987.

[27] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1:134–152, 1997.

[28] Magnus Lindahl, Paul Pettersson, and Wang Yi. Formal Design and Analysis of a Gear-Box Controller. In *Proc. of the 4*th *Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, number 1384 in Lecture Notes in Computer Science, pages 281–297. Springer–Verlag, March 1998.

[29] Henrik Lönn and Paul Pettersson. Formal verification of a TDMA protocol start-up mechanism. In *Proceedings of the 1997 Pacific Rim International Symposium on Fault-Tolerant Systems*, PRFTS '97, pages 235–242, Washington, DC, USA, 1997. IEEE Computer Society.

[30] Tim Regan. Multimedia in temporal lotos: A lip-synchronization algorithm. In *Proceedings of the IFIP TC6/WG6.1 Thirteenth International Symposium on Protocol Specification, Testing and Verification XIII*, pages 127–142, Amsterdam, The Netherlands, The Netherlands, 1993. North-Holland Publishing Co.

[31] Jayme L. Szwarcfiter and Peter E. Lauer. A search strategy for the elementary cycles of a directed graph. *BIT Numerical Mathematics*, 16:192–204, 1976.

[32] Robert E Tarjan. Enumeration of the elementary circuits of a directed graph. Technical report, Department of Computer Science, Cornell University, Ithaca, NY, USA, 1972.

[33] James C. Tiernan. An efficient search algorithm to find the elementary circuits of a graph. *Commun. ACM*, 13:722–726, December 1970.

[34] Stavros Tripakis. Verifying progress in timed systems. In *ARTS'99, 5th International AMAST Workshop on Real-time and Probabilistic Systems*, volume 1601 of *Lecture Notes in Computer Science*, pages 299–314. Springer-Verlag, May 1999.

[35] Stavros Tripakis. Checking timed Büchi automata emptiness efficiently. In *Formal Methods in System Design*, pages 267–292, 2005.

[36] Frits Vaandrager and Adriaan de Groot. Analysis of a biphase mark protocol with UPPAAL and PVS. *Formal Aspects of Computing*, 18:433–458, 2006.

[37] Farn Wang. Model-checking distributed real-time systems with states, events, and multiple fairness assumptions. In Charles Rattray, Savitri Maharaj, and Carron Shankland, editors, *Algebraic Methodology and Software Technology*, volume 3116 of *Lecture Notes in Computer Science*, pages 139–142. Springer Berlin / Heidelberg, 2004.

List of Figures

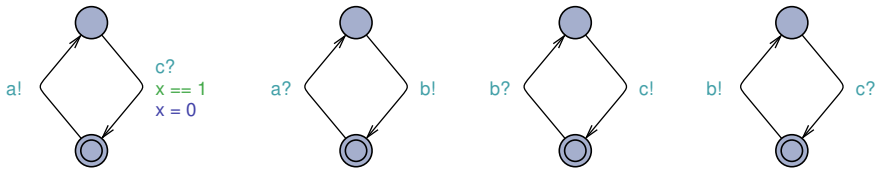
Appendices

A. Test Examples

To give an idea of the tests we used to validate the functionality of ZenoTool we selected four test scenarios from our test suite. One test gives an idea of loop classification and safety propagation, one test shows template parameter binding and safety invalidation by external clock updates, one test tests the transitivity property of the safely dependent data heuristic, and the last test covers the conflicting guard heuristics by checking all possible combinations of guards.

The test suite model is available from the website of the Institute for Software Systems of the Hamburg University of Technology.[1]

A.1. Loop Classification and Safety Propagation



(a) First loop (b) Second loop (c) Third loop (d) Fourth loop

Figure A.1.: Loop classification test involving synchronization

Global declarations

```
chan a, b, c;
```

Local declarations (first loop)

```
clock x;
```

This test features four loops. The first loop is safe by construction due to a local SNZ witness. The second loop becomes safe due to synchronization with the first and third loop. The last two loops could synchronize on their own and thus remain unsafe.

[1] http://www.sts.tu-harburg.de/research/zenotool.html

A.2. Template Parameter Binding and Safety Invalidation

(a) First template (b) Second template

Figure A.2.: Loop classification test involving template parameters

Template parameters (first template)

```
clock & extclock
```

Template parameters (second template)

```
clock & extclock
```

Template instantiation

```
clock externalclock;
Instance1 = FirstTemplate(externalclock);
Instance2 = SecondTemplate(externalclock);
system Instance1, Instance2;
```

This test features two templates that both obtain their clock variable from a template parameter. The clock is provided by reference. The first loop is strongly non-Zeno but its safety depends on external updates on the provided clock variable. In this test the same clock is used by the second template and the second loop provides such an external update. Therefore, both loops are unsafe.

A.3. Transitivity of Safely Dependent Loops

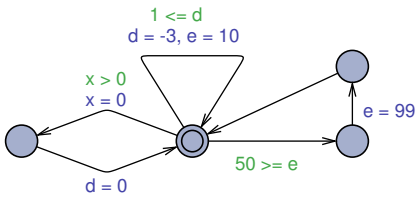


Figure A.3.: Transitivity test for safely dependent loops

Local declarations

```
clock x;
int d, e;
```

This test features a template with three loops. The leftmost loop is safe by construction because of a local SNZ witness. The top loop can only iterate if the safe loop is executed as well, due to its constraints on the data variable `d`. Therefore, it is a safely dependent loop. Then, the right loop also becomes safely dependent as the data update on `e` is safe and there are no other valuations for `e`.

A.4. Conflicting Guards Combination Coverage

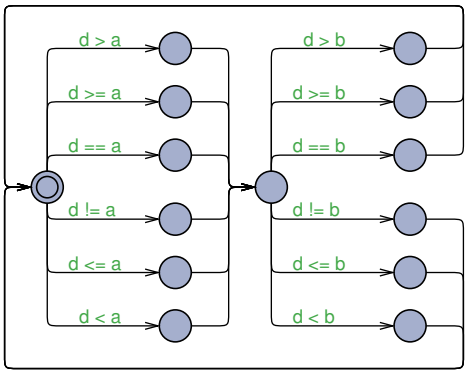


Figure A.4.: Combinations of guards

Local declarations

```
int d;
```

Template parameters

```
const int a, const int b
```

Template instantiation

```
Instance1 = Template(3,5);
Instance2 = Template(5,3);
Instance3 = Template(2,2);
system Instance1, Instance2, Instance3;
```

This test covers all different guard combinations for the determination of conflicting guards. The template features a loop for every combination of two comparison operators. The instantiation ensures all relevant cases for the constants are covered: $a > b$, $a < b$, and $a = b$.

B. Case Study Models

In the following we show three of our analyzed UPPAAL models to give an impression of the appearance of the models and the UPPAAL specification model file format. We selected the models of the Carrier Sense, Multiple-Access with Collision Detection protocol used for UPPAAL benchmarking (see section B.1), the Fischer protocol example (see section B.2), and the Biphase Mark Protocol model (see section B.3) as they show different modeling styles.

All analyzed models can be found on the website of the Institute for Software Systems of the Hamburg University of Technology.[1]

B.1. CSMA/CD Protocol

B.1.1. Declarations

Global declarations

```
chan begin, end, busy, cd1, cd2;
```

Local declarations (bus template)

```
clock x;
```

Local declarations (first participant template)

```
clock x;
```

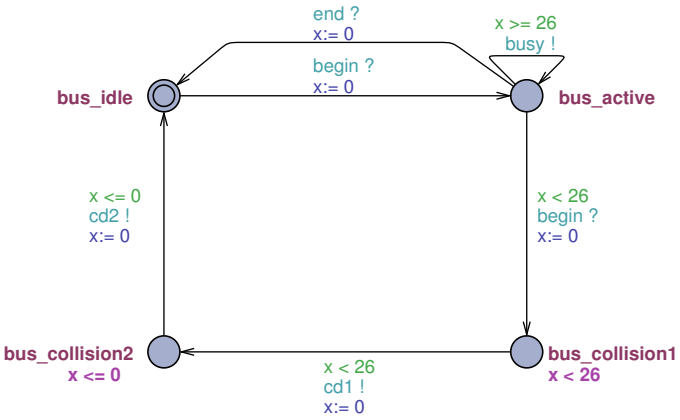
Local declarations (second participant template)

```
clock x;
```

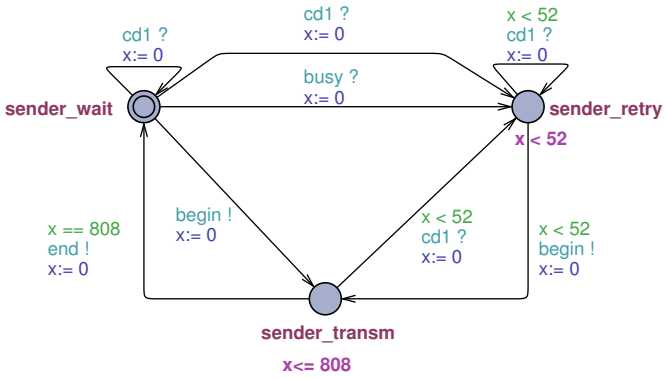
B.1.2. Model File (csmacd2.xml)

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE nta PUBLIC "-//Uppaal Team//DTD Flat System 1.1//EN"
       "http://www.it.uu.se/research/group/darts/uppaal/flat-1_1.dtd">
<nta>
  <declaration>// ----------------------------------------
// CSMA/CD 2
```

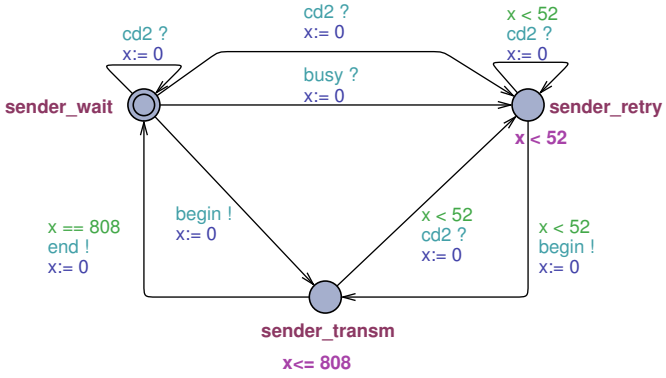
[1] http://www.sts.tu-harburg.de/research/zenotool.html



(a) Bus template



(b) First participant template



(c) Second participant template

Figure B.1.: CSMA/CD protocol model

```
// Carrier Sense, Multiple−Access with Collision Detection
//
// automatically generated by script genCSMA_CD.awk
// M. Oliver Moeller &lt;omoeller@brics.dk&gt;
// Wed Sep 19 11:48:20 2001
// −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
chan begin, end, busy, cd1, cd2;</declaration>
  <template>
    <name>P0</name>
    <declaration>clock x;</declaration>
    <location id="id0" x="-32" y="72">
      <name x="-112" y="64">bus_idle</name>
    </location>
    <location id="id1" x="240" y="72">
      <name x="264" y="64">bus_active</name>
    </location>
    <location id="id2" x="240" y="264">
      <name x="256" y="256">bus_collision1</name>
      <label kind="invariant" x="256" y="272">x &lt; 26</label>
    </location>
    <location id="id3" x="-32" y="264">
      <name x="-152" y="256">bus_collision2</name>
      <label kind="invariant" x="-104" y="272">x &lt;= 0</label>
    </location>
    <init ref="id0"/>
    <transition>
      <source ref="id0"/>
      <target ref="id1"/>
      <label kind="synchronisation" x="80" y="40">begin ?</label>
      <label kind="assignment" x="80" y="56">x:= 0</label>
    </transition>
    <transition>
      <source ref="id1"/>
      <target ref="id0"/>
      <label kind="synchronisation" x="80" y="1">end ?</label>
      <label kind="assignment" x="80" y="16">x:= 0</label>
      <nail x="184" y="32"/>
      <nail x="16" y="32"/>
    </transition>
    <transition>
      <source ref="id1"/>
      <target ref="id1"/>
      <label kind="guard" x="216" y="8">x &gt;= 26</label>
      <label kind="synchronisation" x="224" y="24">busy !</label>
      <nail x="210" y="42"/>
      <nail x="270" y="42"/>
    </transition>
    <transition>
      <source ref="id1"/>
```

```
    <target ref="id2"/>
    <label kind="guard" x="248" y="138">x &lt; 26</label>
    <label kind="synchronisation" x="248" y="153">begin ?</label>
    <label kind="assignment" x="248" y="168">x:= 0</label>
  </transition>
  <transition>
    <source ref="id2"/>
    <target ref="id3"/>
    <label kind="guard" x="88" y="266">x &lt; 26</label>
    <label kind="synchronisation" x="88" y="281">cd1 !</label>
    <label kind="assignment" x="88" y="296">x:= 0</label>
  </transition>
  <transition>
    <source ref="id3"/>
    <target ref="id0"/>
    <label kind="guard" x="-88" y="138">x &lt;= 0</label>
    <label kind="synchronisation" x="-88" y="153">cd2 !</label>
    <label kind="assignment" x="-88" y="168">x:= 0</label>
  </transition>
</template>
<template>
  <name>P1</name>
  <declaration>clock x;</declaration>
  <location id="id4" x="-48" y="88">
    <name x="-152" y="80">sender_wait</name>
  </location>
  <location id="id5" x="88" y="232">
    <name x="40" y="248">sender_transm</name>
    <label kind="invariant" x="56" y="272">x&lt;= 808</label>
  </location>
  <location id="id6" x="240" y="88">
    <name x="256" y="80">sender_retry</name>
    <label kind="invariant" x="230" y="103">x &lt; 52</label>
  </location>
  <init ref="id4"/>
  <transition>
    <source ref="id4"/>
    <target ref="id5"/>
    <label kind="synchronisation" x="-24" y="160">begin !</label>
    <label kind="assignment" x="-24" y="175">x:= 0</label>
  </transition>
  <transition>
    <source ref="id4"/>
    <target ref="id4"/>
    <label kind="synchronisation" x="-64" y="25">cd1 ?</label>
    <label kind="assignment" x="-64" y="40">x:= 0</label>
    <nail x="-78" y="58"/>
    <nail x="-18" y="58"/>
  </transition>
```

```
  <transition>
    <source ref="id4"/>
    <target ref="id6"/>
    <label kind="synchronisation" x="72" y="9">cd1 ?</label>
    <label kind="assignment" x="72" y="24">x:= 0</label>
    <nail x="8" y="48"/>
    <nail x="176" y="48"/>
  </transition>
  <transition>
    <source ref="id4"/>
    <target ref="id6"/>
    <label kind="synchronisation" x="72" y="56">busy ?</label>
    <label kind="assignment" x="72" y="72">x:= 0</label>
  </transition>
  <transition>
    <source ref="id5"/>
    <target ref="id4"/>
    <label kind="guard" x="-120" y="170">x == 808</label>
    <label kind="synchronisation" x="-120" y="185">end !</label>
    <label kind="assignment" x="-120" y="200">x:= 0</label>
    <nail x="-48" y="232"/>
  </transition>
  <transition>
    <source ref="id5"/>
    <target ref="id6"/>
    <label kind="guard" x="160" y="161">x &lt; 52</label>
    <label kind="synchronisation" x="160" y="176">cd1 ?</label>
    <label kind="assignment" x="160" y="191">x:= 0</label>
  </transition>
  <transition>
    <source ref="id6"/>
    <target ref="id5"/>
    <label kind="guard" x="248" y="170">x &lt; 52</label>
    <label kind="synchronisation" x="248" y="185">begin !</label>
    <label kind="assignment" x="248" y="200">x:= 0</label>
    <nail x="240" y="232"/>
  </transition>
  <transition>
    <source ref="id6"/>
    <target ref="id6"/>
    <label kind="guard" x="224" y="10">x &lt; 52</label>
    <label kind="synchronisation" x="224" y="25">cd1 ?</label>
    <label kind="assignment" x="224" y="40">x:= 0</label>
    <nail x="212" y="56"/>
    <nail x="272" y="56"/>
  </transition>
</template>
<template>
  <name>P2</name>
```

```
<declaration>clock x;</declaration>
<location id="id7" x="40" y="80">
  <name x="30" y="50">sender_wait</name>
</location>
<location id="id8" x="190" y="80">
  <name x="180" y="50">sender_transm</name>
  <label kind="invariant" x="180" y="95">x&lt;= 808</label>
</location>
<location id="id9" x="190" y="230">
  <name x="180" y="200">sender_retry</name>
  <label kind="invariant" x="180" y="245">x &lt; 52</label>
</location>
<init ref="id7"/>
<transition>
  <source ref="id7"/>
  <target ref="id8"/>
  <label kind="synchronisation" x="55" y="65">begin !</label>
  <label kind="assignment" x="55" y="80">x:= 0</label>
</transition>
<transition>
  <source ref="id7"/>
  <target ref="id7"/>
  <label kind="synchronisation" x="-20" y="65">cd2 ?</label>
  <label kind="assignment" x="-20" y="80">x:= 0</label>
  <nail x="10" y="50"/>
  <nail x="70" y="50"/>
</transition>
<transition>
  <source ref="id7"/>
  <target ref="id9"/>
  <label kind="synchronisation" x="55" y="140">cd2 ?</label>
  <label kind="assignment" x="55" y="155">x:= 0</label>
</transition>
<transition>
  <source ref="id7"/>
  <target ref="id9"/>
  <label kind="synchronisation" x="55" y="140">busy ?</label>
  <label kind="assignment" x="55" y="155">x:= 0</label>
</transition>
<transition>
  <source ref="id8"/>
  <target ref="id7"/>
  <label kind="guard" x="55" y="50">x == 808</label>
  <label kind="synchronisation" x="55" y="65">end !</label>
  <label kind="assignment" x="55" y="80">x:= 0</label>
</transition>
<transition>
  <source ref="id8"/>
  <target ref="id9"/>
```

```
      <label kind="guard" x="130" y="125">x &lt; 52</label>
      <label kind="synchronisation" x="130" y="140">cd2 ?</label>
      <label kind="assignment" x="130" y="155">x:= 0</label>
    </transition>
    <transition>
      <source ref="id9"/>
      <target ref="id8"/>
      <label kind="guard" x="130" y="125">x &lt; 52</label>
      <label kind="synchronisation" x="130" y="140">begin !</label>
      <label kind="assignment" x="130" y="155">x:= 0</label>
    </transition>
    <transition>
      <source ref="id9"/>
      <target ref="id9"/>
      <label kind="guard" x="130" y="200">x &lt; 52</label>
      <label kind="synchronisation" x="130" y="215">cd2 ?</label>
      <label kind="assignment" x="130" y="230">x:= 0</label>
      <nail x="160" y="200"/>
      <nail x="220" y="200"/>
    </transition>
  </template>
  <system>system P0, P1, P2;</system>
</nta>
```

B.2. Fischer Protocol

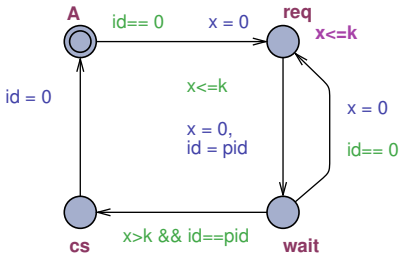


Figure B.2.: Fischer protocol model

B.2.1. Declarations

Global declarations

```
typedef int[1,6] id_t;
int id;
```

Local declarations (protocol template)

```
clock x;
const int k = 2;
```

Template parameters (protocol template)

```
const id_t pid
```

B.2.2. Model File (fischer6.xml)

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE nta PUBLIC "-//Uppaal Team//DTD Flat System 1.1//EN"
       "http://www.it.uu.se/research/group/darts/uppaal/flat-1_1.dtd">
<nta>
  <declaration>// Mutual exclusion protocol by Fischer.

typedef int[1,6] id_t;
int id;</declaration>
  <template>
    <name x="16" y="-8">P</name>
    <parameter>const id_t pid</parameter>
    <declaration>clock x;
const int k = 2;</declaration>
    <location id="id0" x="216" y="176">
      <name x="216" y="192">wait</name>
    </location>
    <location id="id1" x="216" y="48">
      <name x="216" y="16">req</name>
      <label kind="invariant" x="240" y="32">x&lt;=k</label>
    </location>
    <location id="id2" x="64" y="48">
      <name x="54" y="18">A</name>
    </location>
    <location id="id3" x="64" y="176">
      <name x="56" y="192">cs</name>
    </location>
    <init ref="id2"/>
    <transition>
      <source ref="id2"/>
      <target ref="id1"/>
      <label kind="guard" x="88" y="24">id== 0</label>
      <label kind="assignment" x="160" y="24">x = 0</label>
    </transition>
    <transition>
      <source ref="id1"/>
      <target ref="id0"/>
      <label kind="guard" x="144" y="72">x&lt;=k</label>
      <label kind="assignment" x="144" y="104">x = 0,
id = pid</label>
    </transition>
    <transition>
      <source ref="id0"/>
      <target ref="id1"/>
```

```
      <label kind="guard" x="264" y="120">id== 0</label>
      <label kind="assignment" x="264" y="88">x = 0</label>
      <nail x="251" y="146"/>
      <nail x="251" y="82"/>
    </transition>
    <transition>
      <source ref="id0"/>
      <target ref="id3"/>
      <label kind="guard" x="96" y="184">x&amp;gt;k &amp;amp;&amp;amp; id==pid</label>
    </transition>
    <transition>
      <source ref="id3"/>
      <target ref="id2"/>
      <label kind="assignment" x="8" y="80">id = 0</label>
    </transition>
  </template>
  <system>system P;</system>
</nta>
```

B.3. Biphase Mark Protocol

B.3.1. Declarations

Global declarations

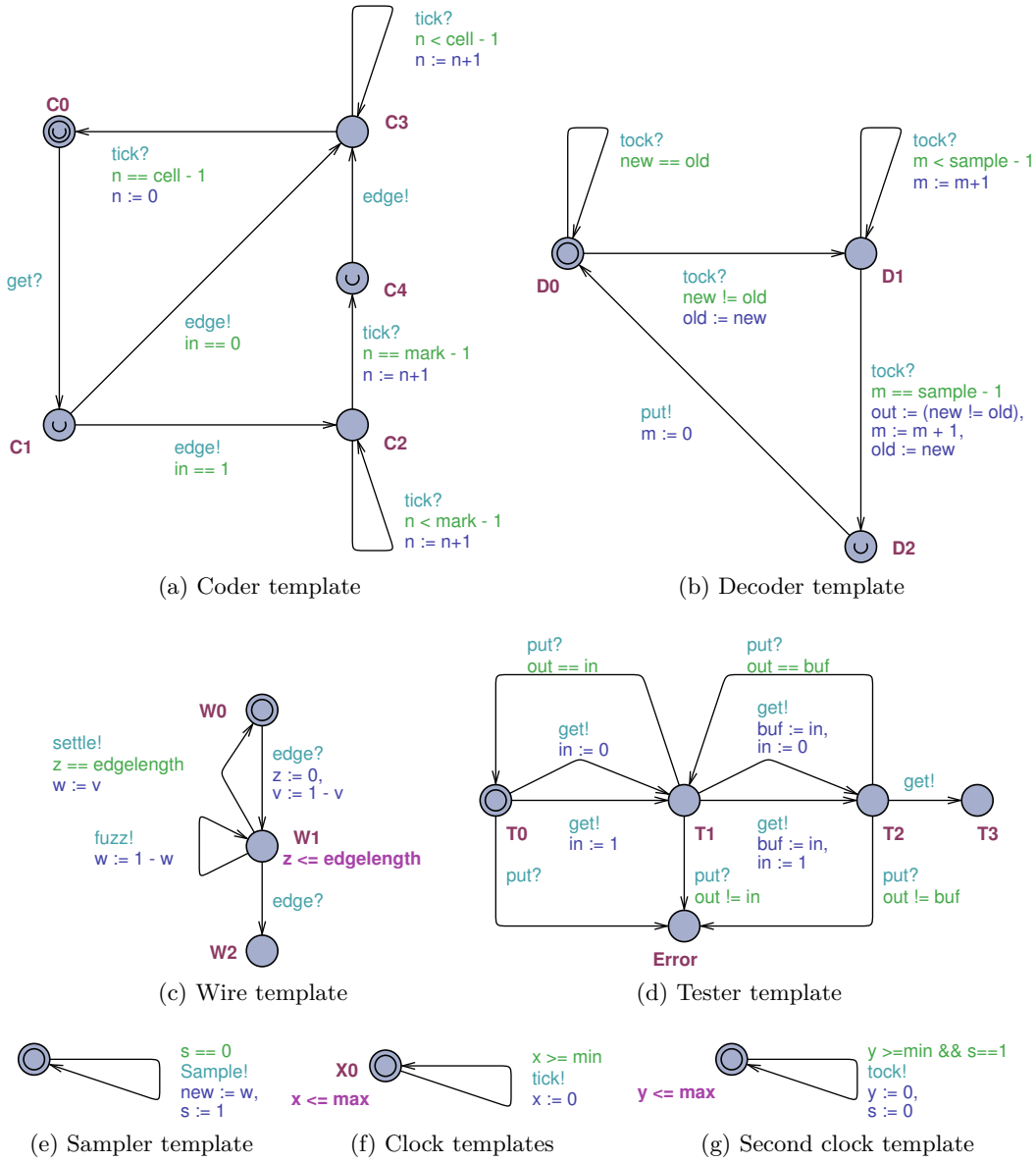
```
const int cell = 14;
const int mark = 7;
const int sample = 10;
const int min = 93;
const int max = 100;
const int edgelength = 100;

chan tick, tock, edge, get, put;
broadcast chan fuzz, settle, Sample;
int m, n;
bool in, out, v, w, s, new, old, buf;
clock x, y, z;
```

B.3.2. Model File (bmp.xml)

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE nta PUBLIC "-//Uppaal Team//DTD Flat System 1.1//EN"
       "http://www.it.uu.se/research/group/darts/uppaal/flat-1_1.dtd">
<nta>
  <declaration>// Global Declarations

chan tick, tock, edge, get, put;
```



(a) Coder template (b) Decoder template

(c) Wire template (d) Tester template

(e) Sampler template (f) Clock templates (g) Second clock template

Figure B.3.: Biphase Mark protocol model

```
broadcast chan fuzz, settle, Sample;

int m, n;

bool in, out, v, w, s, new, old, buf;

clock x, y, z;

const int cell = 14;

const int mark = 7;

const int sample = 10;

const int min = 93;

const int max = 100;

const int edgelength = 100;</declaration>
  <template>
    <name x="5" y="5">Coder</name>
    <location id="id0" x="320" y="240">
      <name x="344" y="240">C4</name>
      <urgent/>
    </location>
    <location id="id1" x="320" y="128">
      <name x="345" y="112">C3</name>
    </location>
    <location id="id2" x="320" y="352">
      <name x="344" y="357">C2</name>
    </location>
    <location id="id3" x="96" y="352">
      <name x="58" y="360">C1</name>
      <urgent/>
    </location>
    <location id="id4" x="96" y="128">
      <name x="86" y="98">C0</name>
      <urgent/>
    </location>
    <init ref="id4"/>
    <transition>
      <source ref="id4"/>
      <target ref="id3"/>
      <label kind="synchronisation" x="56" y="232">get?</label>
    </transition>
    <transition>
      <source ref="id3"/>
      <target ref="id2"/>
      <label kind="guard" x="184" y="376">in == 1</label>
```

```
    <label kind="synchronisation" x="184" y="360">edge!</label>
  </transition>
  <transition>
    <source ref="id2"/>
    <target ref="id2"/>
    <label kind="guard" x="360" y="416">n &lt; mark − 1</label>
    <label kind="synchronisation" x="360" y="400">tick?</label>
    <label kind="assignment" x="360" y="432">n := n+1</label>
    <nail x="320" y="448"/>
    <nail x="352" y="448"/>
  </transition>
  <transition>
    <source ref="id3"/>
    <target ref="id1"/>
    <label kind="guard" x="192" y="280">in == 0</label>
    <label kind="synchronisation" x="192" y="264">edge!</label>
  </transition>
  <transition>
    <source ref="id1"/>
    <target ref="id1"/>
    <label kind="guard" x="368" y="48">n &lt; cell − 1</label>
    <label kind="synchronisation" x="368" y="32">tick?</label>
    <label kind="assignment" x="368" y="64">n := n+1</label>
    <nail x="320" y="32"/>
    <nail x="352" y="32"/>
  </transition>
  <transition>
    <source ref="id1"/>
    <target ref="id4"/>
    <label kind="guard" x="136" y="152">n == cell − 1</label>
    <label kind="synchronisation" x="136" y="136">tick?</label>
    <label kind="assignment" x="136" y="168">n := 0</label>
  </transition>
  <transition>
    <source ref="id0"/>
    <target ref="id1"/>
    <label kind="synchronisation" x="328" y="168">edge!</label>
  </transition>
  <transition>
    <source ref="id2"/>
    <target ref="id0"/>
    <label kind="guard" x="328" y="288">n == mark − 1</label>
    <label kind="synchronisation" x="328" y="272">tick?</label>
    <label kind="assignment" x="328" y="304">n := n+1</label>
  </transition>
</template>
<template>
  <name x="5" y="5">Clock</name>
  <location id="id5" x="128" y="96">
```

```
      <name x="90" y="90">X0</name>
      <label kind="invariant" x="56" y="112">x &lt;= max</label>
    </location>
    <init ref="id5"/>
    <transition>
      <source ref="id5"/>
      <target ref="id5"/>
      <label kind="guard" x="240" y="80">x &gt;= min</label>
      <label kind="synchronisation" x="240" y="96">tick!</label>
      <label kind="assignment" x="240" y="112">x := 0</label>
      <nail x="224" y="128"/>
      <nail x="224" y="96"/>
    </transition>
  </template>
  <template>
    <name x="8" y="0">Wire</name>
    <location id="id6" x="544" y="128">
      <name x="534" y="98">W2</name>
    </location>
    <location id="id7" x="352" y="128">
      <name x="384" y="136">W1</name>
      <label kind="invariant" x="384" y="152">z &lt;= edgelength</label>
    </location>
    <location id="id8" x="160" y="128">
      <name x="112" y="120">W0</name>
    </location>
    <init ref="id8"/>
    <transition>
      <source ref="id7"/>
      <target ref="id7"/>
      <label kind="synchronisation" x="392" y="200">fuzz!</label>
      <label kind="assignment" x="392" y="216">w := 1 - w</label>
      <nail x="352" y="224"/>
      <nail x="384" y="224"/>
    </transition>
    <transition>
      <source ref="id8"/>
      <target ref="id7"/>
      <label kind="synchronisation" x="192" y="72">edge?</label>
      <label kind="assignment" x="192" y="88">z := 0,
v := 1 - v</label>
    </transition>
    <transition>
      <source ref="id7"/>
      <target ref="id8"/>
      <label kind="guard" x="176" y="208">z == edgelength</label>
      <label kind="synchronisation" x="176" y="192">settle!</label>
      <label kind="assignment" x="176" y="224">w := v</label>
      <nail x="220" y="192"/>
```

```
    </transition>
    <transition>
      <source ref="id7"/>
      <target ref="id6"/>
      <label kind="synchronisation" x="456" y="104">edge?</label>
    </transition>
  </template>
  <template>
    <name x="5" y="5">Sampler</name>
    <location id="id9" x="96" y="128"/>
    <init ref="id9"/>
    <transition>
      <source ref="id9"/>
      <target ref="id9"/>
      <label kind="guard" x="208" y="112">s == 0</label>
      <label kind="synchronisation" x="208" y="128">Sample!</label>
      <label kind="assignment" x="208" y="144">new := w,
s := 1</label>
      <nail x="192" y="128"/>
      <nail x="192" y="160"/>
    </transition>
  </template>
  <template>
    <name x="5" y="5">Decoder</name>
    <location id="id10" x="320" y="384">
      <name x="344" y="376">D2</name>
      <urgent/>
    </location>
    <location id="id11" x="320" y="160">
      <name x="336" y="168">D1</name>
    </location>
    <location id="id12" x="96" y="160">
      <name x="72" y="176">D0</name>
    </location>
    <init ref="id12"/>
    <transition>
      <source ref="id12"/>
      <target ref="id11"/>
      <label kind="guard" x="184" y="184">new != old</label>
      <label kind="synchronisation" x="184" y="168">tock?</label>
      <label kind="assignment" x="184" y="200">old := new</label>
    </transition>
    <transition>
      <source ref="id10"/>
      <target ref="id12"/>
      <label kind="synchronisation" x="152" y="272">put!</label>
      <label kind="assignment" x="152" y="288">m := 0</label>
    </transition>
    <transition>
```

```
      <source ref="id11"/>
      <target ref="id10"/>
      <label kind="guard" x="328" y="256">m == sample − 1</label>
      <label kind="synchronisation" x="328" y="240">tock?</label>
      <label kind="assignment" x="328" y="272">out := (new != old),
m := m + 1,
old := new</label>
    </transition>
    <transition>
      <source ref="id11"/>
      <target ref="id11"/>
      <label kind="guard" x="360" y="80">m &lt; sample − 1</label>
      <label kind="synchronisation" x="360" y="64">tock?</label>
      <label kind="assignment" x="360" y="96">m := m+1</label>
      <nail x="320" y="64"/>
      <nail x="352" y="64"/>
    </transition>
    <transition>
      <source ref="id12"/>
      <target ref="id12"/>
      <label kind="guard" x="136" y="80">new == old</label>
      <label kind="synchronisation" x="136" y="64">tock?</label>
      <nail x="96" y="64"/>
      <nail x="128" y="64"/>
    </transition>
  </template>
  <template>
    <name x="8" y="0">Tester</name>
    <location id="id13" x="576" y="224">
      <name x="576" y="240">T3</name>
    </location>
    <location id="id14" x="416" y="224">
      <name x="424" y="240">T2</name>
    </location>
    <location id="id15" x="256" y="224">
      <name x="264" y="240">T1</name>
    </location>
    <location id="id16" x="256" y="384">
      <name x="232" y="400">Error</name>
    </location>
    <location id="id17" x="96" y="224">
      <name x="104" y="240">T0</name>
    </location>
    <init ref="id17"/>
    <transition>
      <source ref="id17"/>
      <target ref="id15"/>
      <label kind="synchronisation" x="152" y="232">get!</label>
      <label kind="assignment" x="152" y="248">in := 1</label>
```

```
    </transition>
    <transition>
      <source ref="id15"/>
      <target ref="id14"/>
      <label kind="synchronisation" x="312" y="232">get!</label>
      <label kind="assignment" x="312" y="248">buf := in,
in := 1</label>
    </transition>
    <transition>
      <source ref="id15"/>
      <target ref="id16"/>
      <label kind="guard" x="264" y="336">out != in</label>
      <label kind="synchronisation" x="264" y="320">put?</label>
    </transition>
    <transition>
      <source ref="id17"/>
      <target ref="id16"/>
      <label kind="synchronisation" x="104" y="320">put?</label>
      <nail x="96" y="384"/>
    </transition>
    <transition>
      <source ref="id17"/>
      <target ref="id15"/>
      <label kind="synchronisation" x="152" y="160">get!</label>
      <label kind="assignment" x="152" y="176">in := 0</label>
      <nail x="160" y="192"/>
    </transition>
    <transition>
      <source ref="id15"/>
      <target ref="id17"/>
      <label kind="guard" x="120" y="80">out == in</label>
      <label kind="synchronisation" x="120" y="64">put?</label>
      <nail x="224" y="96"/>
      <nail x="96" y="96"/>
    </transition>
    <transition>
      <source ref="id15"/>
      <target ref="id14"/>
      <label kind="synchronisation" x="312" y="144">get!</label>
      <label kind="assignment" x="312" y="160">buf := in,
in := 0</label>
      <nail x="320" y="192"/>
    </transition>
    <transition>
      <source ref="id14"/>
      <target ref="id15"/>
      <label kind="guard" x="312" y="80">out == buf</label>
      <label kind="synchronisation" x="312" y="64">put?</label>
      <nail x="416" y="96"/>
```

```
      <nail x="288" y="96"/>
    </transition>
    <transition>
      <source ref="id14"/>
      <target ref="id16"/>
      <label kind="guard" x="424" y="336">out != buf</label>
      <label kind="synchronisation" x="424" y="320">put?</label>
      <nail x="416" y="384"/>
    </transition>
    <transition>
      <source ref="id14"/>
      <target ref="id13"/>
      <label kind="synchronisation" x="472" y="232">get!</label>
    </transition>
  </template>
  <template>
    <name x="5" y="5">Clock2</name>
    <location id="id18" x="128" y="128">
      <label kind="invariant" x="56" y="144">y &lt;= max</label>
    </location>
    <init ref="id18"/>
    <transition>
      <source ref="id18"/>
      <target ref="id18"/>
      <label kind="guard" x="232" y="112">y &gt;=min &amp;&amp; s==1</label>
      <label kind="synchronisation" x="232" y="128">tock!</label>
      <label kind="assignment" x="232" y="144">y := 0,
s := 0</label>
      <nail x="224" y="128"/>
      <nail x="224" y="160"/>
    </transition>
  </template>
  <system>
system Coder, Clock, Wire, Sampler, Clock2, Decoder, Tester;</system>
</nta>
```