Technische Universität Hamburg-Harburg

Bachelor Thesis

Detection of Zeno Sets in Hybrid Systems to Validate Modelica Simulations

Marcel Gehrke

July 20, 2012

supervised by:
Prof. Dr. Sibylle Schupp

Technische Universität Hamburg-Harburg
Institute for Software Systems
Schwarzenbergstraße 95
21073 Hamburg

Eidesstattliche Erklärung

Ich versichere an Eides statt, dass ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit wurde in dieser oder ähnlicher Form noch keiner Prüfungskommission vorgelegt.

Hamburg, den 20. Juli 2012

Marcel Gehrke

Acknowledgments

I would like to thank Prof. Dr. Sibylle Schupp for the opportunity to write this thesis. I am also very thankful for her valuable feedback and good supervision of my thesis development.

Contents

List of Figures

List of Tables

1 Introduction

Throughout this thesis, we will focus on hybrid systems [12]. Hybrid systems are interactive systems and combine both continuous and discrete state dynamics [15]. Interactive systems react to events triggered by the environment [8]. While reacting to the event, the system might even change the environment.

A simple example of a hybrid system is an air conditioning system. It consists of a heater and a sensor for the current temperature. Based on the current temperature, the heater is either on or off. Hence, the system is interacting with its environment, namely the temperature.

Hybrid systems are rather sensitive systems, in the way that a failure of such system could be critical. One only has to think of a power plant. In a power plant many components need to be cooled. Water can be used for the purpose of cooling. The water could be stored in a water tank. Building the system and testing the system by running it, is not a good idea, since if we do not always have enough water in the tanks a possible outcome would be a meltdown. Hence, it is crucial to validate that there is always enough water in the tank.

Such a water tank system is a hybrid system. In the water tank, we have sensors measuring the current water level. If the water level drops below a desired level, a source pours water at a constant rate into the tank. Thus, the system is interacting with its environment, since it reacts to an event triggered by the environment and even changes it by reacting on the event. A way to validate a hybrid system is simulation. During simulation, it is possible to test the system. A tool based on the Modelica language could be used for simulating hybrid systems [5].

However, to simulate a hybrid system, we first have to model the system. While modeling the system, we need to abstract the hybrid system. Unfortunately, the process of abstraction can be error prone. Mostly, these errors are due to mathematical complexity and oversimplification [16],[11]. Therefore, the outcome of a simulation of a system can differ from reality [3].

In case a model of a system shows zeno behavior, the outcome of the simulation and the physical system can differ. Zeno behavior can only occur in a model of hybrid [10] or temporal [17] systems, but not in the physical system. Roughly speaking, a hybrid

system shows zenoness if it undergoes an unbounded number of discrete transitions in a finite and bounded length of time [10].

Here, my work comes into play. I implemented a tool to detect zeno sets. The tool tells the user when the model of a system can exhibits zeno behavior. Even more, in case the model can show zeno behavior, the zeno set of that model is printed out. The zeno set contains all points to which the simulation can converge while showing zeno behavior. If the simulation converges to one of these points, the modeler of the simulation should check whether the outcome is trustworthy. Hence, one more safety net for validating a hybrid system is generated. Bauer [2] describes another approach to validating hybrid systems.

The thesis consists of the following parts: In Chapter 2, we take a closer look at hybrid systems and explain them in more detail. I present the simulation tool OpenModelica, which is based on the Modelica language, in Chapter 3. Further, I demonstrate how a simulation can differ from the real world. In Chapter 4, we discuss how it may occur, that a simulation differs from the real world. Therefore, we need a deeper understanding of how zeno behavior occurs. Hence, we focus mainly on understanding zeno behavior in Chapter 4. In Chapter 5, my contribution starts. Here, I present the tool to detect zeno sets, describing what the tool does and how it works. In Chapter 6, I conduct an evaluation of the tool. For the evaluation of the tool, I show that it works for examples given throughout the thesis, as well as a completeness evaluation. Finally, future work and how one could deal with hybrid systems showing zeno behavior is described in Chapter 7.

2 Hybrid Systems

Hybrid systems are systems that are coupled with the environment. Areas such as aeronautics, automotive vehicles, bioengineering, embedded software, process control, and transportation use hybrid systems. Due to their increasing importance, these systems become of more and more interest [12].

Hybrid systems combine both continuous and discrete state dynamics. A state variable is discrete in case it can just take a finite number of values. A continuous state variable can take any value form the Euclidean space $\mathbb{R}^n$ [14]. The change of the continuous variables is normally represented as a differential equation. Throughout the thesis, we abbreviate discrete state by state.

A formal representation of hybrid systems are hybrid automata. We take a closer look at hybrid automata as representation of hybrid systems, in this thesis.

2.1 Example: Air Conditioning

As mentioned in Chapter 1, an example of a hybrid system is an air conditioning system. It can be modeled using two discrete states, on and off, as well as one continuous variable, the current temperature.

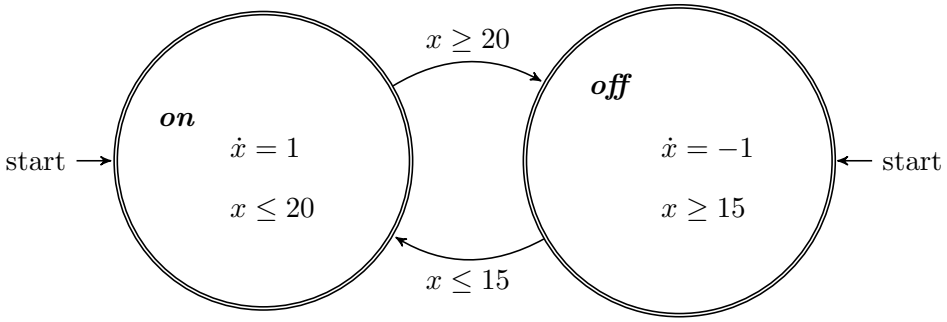

Figure 2.1: Hybrid Automaton of an Air Conditioning as a Heater

Fig. 2.1 shows an air conditioning system as a heater modeled as a hybrid automaton. In the modeled air conditioning system, the current temperature always stays between $15°C$ and $20°C$, as long as the outside temperature is below $15°C$.

Still, we should check whether the temperature really always stays between $15°C$ and $20°C$. Therefore, let us assume the current temperature is $17°C$ in the beginning. Further assume the air conditioning system starts in the *on* state. Hence, the system starts to heat the room. When the current temperature reaches $20°C$, the transition to the *off* state triggers and the automaton switches to the *off* state. In the *off* state, the system stops to heat the room and the room starts to cool down again. When the current temperature drops to $15°C$, the transition to the *on* state is triggered and the process starts all over again.

Thus, modeling the air conditioning system in such a manner guarantees the temperature to always stay between $15°C$ and $20°C$.

2.2 Hybrid Automata

A hybrid automaton is a dynamical system describing the evolution of a set of discrete and continuous variables over time [15].

Definition 1. *A hybrid automaton H is a 7-tupel $H = (Q, X, Init, D, E, G, R)$: [20]*

- *Q, denotes the discrete states: $\{q_1, .., q_n\}$*
- *X, denotes the continuous variables from $\mathbb{R}^n$ with $n \geq 1$*
- *$Init$, denotes the initial states from Q*
- *D: $Q \rightarrow \mathcal{P}(X)$, denotes the domain, the legal region of operation, for each discrete state*
- *$E \subset Q \times Q$, denotes the edge or transition from one state to another: $q_i \rightarrow q_j$*
- *Guards G : $E \rightarrow \mathcal{P}(X)$, denotes a mapping from a transition to a guard. When the guard holds true, the corresponding transition can be triggered.*
- *Reset map R : $E \times X \rightarrow \mathcal{P}(X)$, determines how continuous variables change during a discrete state transition*

In some papers, the domain is called invariant of the state.

2.2.1 General Description

The set Q consists of all states. We only consider automata where every state contains at least one differential equation, ensuring that we do not have any state in which the system idles. Further, we operate in every state on at least one continuous variable. Every continuous variable from the set X has to be in a differential equation in at least one state. Hereby, we ensure that the value of every continuous variable can change over time. To determine in which state the automaton can start, the $Init$ set is crucial. When starting the automaton, the automaton can only be in one of the states defined by the $Init$ set. The $Init$ set is indicated by a double cycle in the hybrid automaton.

Furthermore, every state has a domain, its legal region of operation. A state can only operate as long as the domain of the state holds true. The set E consists of all edges, so called transitions, from one state to another. Every transition has a corresponding guard. The guard declares when a transition can be triggered, by defining the legal region of operation for the transition. Another component of the transition is the reset map. The reset map determines how continuous variables change while taking the corresponding transition. Examples of the use of the reset map are a damping coefficient or simulating a delay.

2.2.2 Air Conditioning

The hybrid automaton in Fig. 2.1 contains all of the elements of Definition 1. The set of discrete states $Q = \{on, off\}$ are the states the automaton can operate in. The set of continuous variables $X = \{x\}$ contains only one element, which represents the current temperature. Even though there are two distinct states, *on* and *off*, the set of continuous variables X is just from $\mathbb{R}^1$. Both states operate on the same continuous variable.

The set of initial states contains the *on* and *off* state, meaning the automaton can start in either of these states. The *on* state has the domain $x \leq 20$. Therefore, the automaton can only operate in the *on* state until the current temperature reaches $20°C$. The domain of the *off* state is $x \geq 15$. While being in the *off* state, the current temperature can only cool down to $15°C$.

The set E of the hybrid automaton consists of two transitions, one transition from *on* to *off* and another one from *off* to *on*. Each of these transitions has a corresponding guard and reset map. The guard for the transition from the *on* to the *off* state, $x \geq 20$, specifies that the transition can be triggered in case the current temperature is at least $20°C$. Comparing the guard with the domain of the *on* state, one can see

that the transition is triggered exactly when the current temperature reaches $20°C$. The *on* state can warm up the temperature until it reaches $20°C$ and the transition can be triggered when the temperature is at least $20°C$. Thus, the only intersection of the domain and guard is when the temperature is exactly $20°C$.

The transition from the *off* to the *on* state adheres to the guard $x \leq 15$. Therefore, the transition can be triggered when the current temperature is $15°C$ or below. However, comparing the guard with the domain of the *off* state yields that the transition can only be triggered when the current temperature is exactly $15°C$, using the same argument as before. The reset map for both transitions is 1, since the continuous variables do not change during taking a transition.

2.3 Example: Water Tank

We will now introduce another example of a hybrid system, a water tank system. The example of the water tank system will follow us throughout the thesis to explain what zeno behavior is.

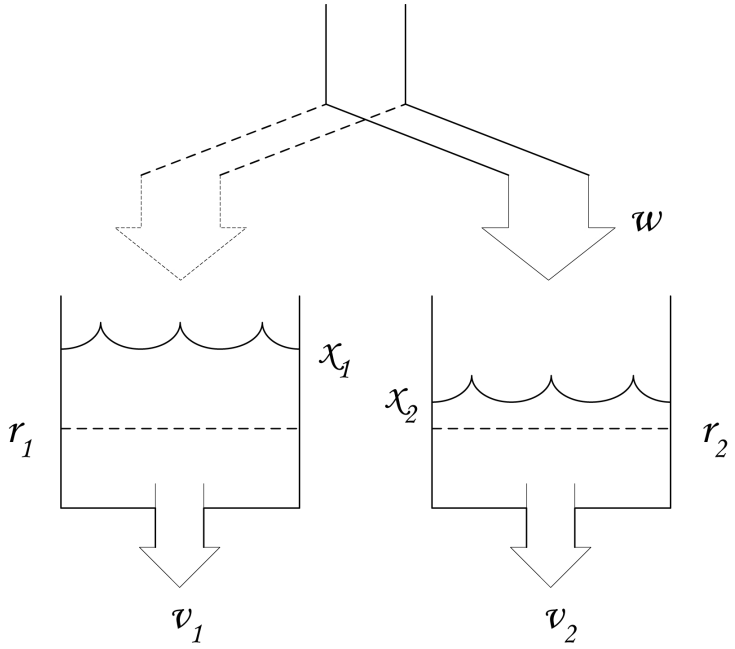

Figure 2.2: Water Tank as Graphical Representation

Fig. 2.2 shows the water tank system of Alur and Henzinger [1]. The water tank system consists of two water tanks, a water source and a hole in each tank. From the hole in each tank, water drains with a constant rate v_1 and v_2 respectively. The water source can only pour water, at a constant rate w, into one tank at a time. Hence, only one tank receives water from the source, while both are draining water at the same time.

Furthermore, the variables x_1 and x_2 in the Fig. 2.2 represent the current water level, while the variable r stands for the minimal required water level. To be able to simulate the water tank system, we need an accurate model of it. Therefore, in Fig. 2.3, we can see one approach to model the water tank system as a hybrid automaton.

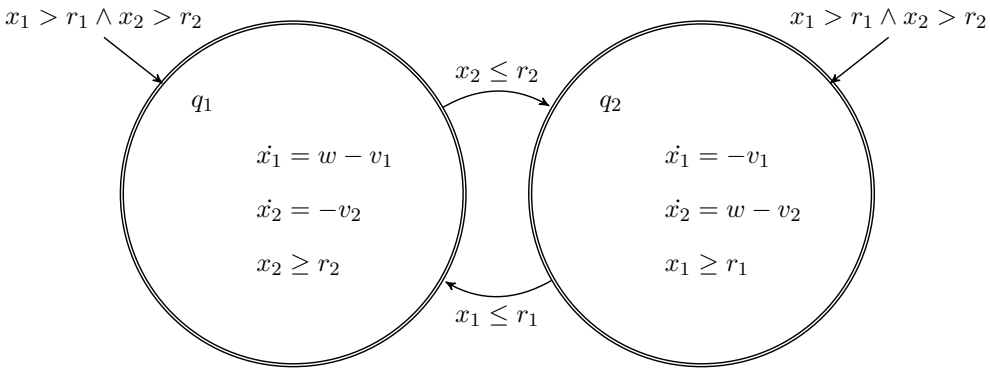

Figure 2.3: Hybrid Automaton of the Water Tank

The automaton has two discrete states denoted by the variables q_1 and q_2. They seem to be related to the two water tanks. More accurately said, the states represent that the water source w pours water into either of the tanks. Both states have a corresponding domain: $x_2 \geq r_2$ for state q_1 and $x_1 \geq r_1$ for q_2. The set X consists of the variables x_1 and x_2. The two transitions, one from q_1 to q_2 and another one from q_2 to q_1, build the elements of the set E. Each of the transitions has a corresponding guard and reset map. The transition from q_1 to q_2 has the guard $x_2 \leq r_2$ and the transition from q_2 to q_1 $x_1 \leq r_1$. The reset map is equal to 1 in both cases. Now, let us check whether the hybrid automaton really reflects the water tank system from Fig. 2.2.

In state q_1, the differential equation $\dot{x_1} = w - v_1$ reflects the case of the water source being above the first tank and pouring water, for every time interval, with the constant rate of w into it. Water drains with a rate of v_1 from the water tank. The case that water only drains from the hole in the second state, is denoted by the differential equation $\dot{x_2} = -v_2$. The domain of q_1, $x_2 \geq r_2$, defines that the

automaton can only operate in the state q_1 as long as the current water level in the second tank is above or equal to the minimal required water level r_2. There is one outgoing transition from q_1 to q_2 with the guard $x_2 \leq r_2$. Due to the guard, the transition can only be triggered when the current water level in the second tank is equal or below the minimal required water level r_2. Comparing the domain of q_1 with the guard from the transition from q_1 to q_2, one can see that the only possibility for the transition to be triggered is when the current water level in the second tank is equal to the minimal required water level, i.e. $x_2 = r_2$.

So far, the model of the water tank system seems to be reasonable. The state q_2 is similar to the state q_1. The differential equations $\dot{x_1} = -v_1$ and $\dot{x_2} = w - v_2$ represent the reversed case of the differential equations from the state q_1. While the water source pours water at a constant rate of w into the second water tank, water drains from both tanks with a constant rate of v_1 and v_2 respectively. The domain $x_1 \geq r_1$ has the same impact as the one from state q_1. Namely, that the automaton can only operate in q_2 as long as there is enough water in the first tank. Finally, there is the outgoing transition to q_1 with the guard $x_1 \leq r_1$. The guard can be triggered when the current water level x_1 is equal or below the minimal required water level r_1. Comparing the domain and the guard, the transition can only be triggered in case of $x_1 = r_1$.

In the next chapter, we will take a look at a simulation tool for hybrid systems. Further, we will see how the water tank example is modeled in that tool.

3 Modelica

Many simulation and modeling tools are available. However, just a few fit our purpose of simulating hybrid systems. Modelica, gPROMS, AnyLogic, VHDL-AMS and Verilog-AMS simulators could be used for simulating hybrid systems [5]. We will focus on tools based on the Modelica language. Modelica is a modern, strongly typed, declarative, and object-oriented language for modeling and simulation of complex systems [6]. The Modelica langauge differs in two important aspect from other object-oriented language like Java or C++.

The first aspect is that Modelica is a modeling language. Unlike a programming language, a Modelica model is not compiled in the usual sense but is translated into objects that are then interpreted by a simulation engine. A Modelica model is primarily a declarative mathematical description that simplifies further analysis. The second and more important difference, for our purpose, is that Modelica focuses on differential equations. The fact that Modelica focuses on differential equations makes Modelica well suited for simulating hybrid systems [7].

There are many implementations of the Modelica language, such as AMESim, CATIA Systems, Dymola, JModelica.org, MapleSim, Wolfram SystemModeler, OpenModelica, Scicos, SimulationX, Vertex and Xcos [1]. We will focus on the open-source tool OpenModelica.

3.1 OpenModelica

OpenModelica is a frontend for the Modelica language. It can be used to simulate Modelica models and therefore, hybrid systems. Further, the editor of OpenModelica provides an efficient environment to write Modelica models. In OpenModelica, the Modelica models are translated into C code using a compiler, and then evaluated.

Translating the Modelica model to C code has one major advantage: It provides the possibility to have access to a debugger. The OpenModelica debugger makes

[1] `https://www.modelica.org/tools`

OpenModelica unique compared to other Modelica frontends since it is the only one so far with good support for debugging Modelica algorithmic code [5].

Being an open-source tool makes OpenModelica rather transparent compared to other Modelica frontends. Thus, it is possible to modify OpenModelica in a desired way. It is also possible to write own libraries. One of these user-written libraries is the hybrid automata library (HyAuLib). Unfortunately, the hybrid automata library is not maintained anymore and therefore, hardly supported. However, the hybrid automata library helps us modeling our hybrid systems. Hence, we will introduce it now.

3.1.1 Hybrid Automata Library

OpenModelica is already well suited for simulating hybrid systems [4]. However, using the hybrid automata library, it becomes more intuitive to model hybrid systems in OpenModelica. Hybrid systems are formally represented as hybrid automata. Therefore, using the representation as automata for writing models of hybrid systems for simulations is more natural.

The library has been derived by extending the free Modelica StateGraph library by Otter and Dressler [18], which is based on the JGraphChart method and provides components to model finite state machines [15]. Instead of having to write the explicit Modelica code, which can be burdensome and error-prone even for very simple models, the hybrid automata paradigm can now be used in a natural fashion. The HyAuLib takes care of all necessary state transitions.

There are two basic elements of automata, which are modeled in the HyAuLib, the *FiniteStates* and the *Transitions*.

- *FiniteStates:*
  Important elements of the state are the continuous dynamics as well as the state domain or invariant. These two elements also have to be defined. Every state can have a defined number of incoming and outgoing connections.

- *Transitions:*
  The *Transitions* contain two elements. One element is the guard. Every transition in a hybrid automaton has to have a corresponding guard, which determines when a transition is triggered. The other element is a reset map. There are two possible reset conditions available. The standard reset map is 1, which is the first possibility. The second possibility allows the users to specify their own reset map.

Having all of these elements it is possible to model a hybrid automata in OpenModelica using HyAuLib. Further, one can specify a transition delay, meaning that a transition is not instantly triggered when the guard holds true, but after a specified delay.

3.1.2 Bouncing Ball

The bouncing ball is a example used rather often in the field of hybrid systems. It can be modeled as a hybrid automaton as shown in Fig. 3.1.

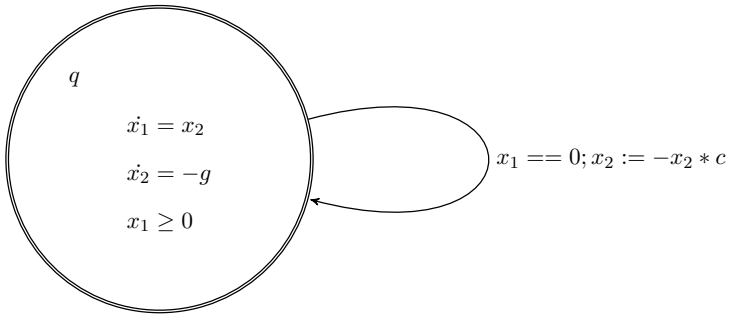

Figure 3.1: Hybrid Automaton of the Bouncing Ball

To model the bouncing ball, only one state is needed. The continuous variable x_1 represents the current height of the ball, while x_2 models the velocity. The invariant of the state, $x_1 \geq 0$, determines that the ball cannot go below the surface. In addition to that, the transition is triggered when the ball hits the ground. Further, the reset map states that in this case, x_2, the velocity gets reversed and multiplied with a variable $c \in [0, 1)$, which represents the damping coefficient.

Obviously, one can also write a Modelica flat model to realize such a bouncing ball. However, the zeno behavior of the bouncing ball system causes some problems here. Due to numerical errors, x_1 will eventually become negative and, since the equations used to describe the model are still satisfied, the ball position will keep decreasing [15]. In the next chapter, we will see what zeno behavior is and how the numerical errors occur. The invariant set of the hybrid automaton using HyAuLib solves this problem, by marking negative values for x_1 as infeasible.

Therefore, HyAuLib can reduce the problems of numerical errors introduced by zeno behavior. Hence, modeling hybrid systems using the HyAuLib is more intuitive and less error prone than writing Modelica flat code.

3.2 Water Tank in OpenModelica

Let us revisit the water tank system from the Fig. 2.2 and model it in OpenModelica. Fig. 3.2 shows how the water tank could be modeled using OpenModelica.

```
model WaterTank
  parameter Real w = 1.8;        //Water from the source
  parameter Real v_1 = 1;        //Water drainage in the first tank
  parameter Real v_2 = 1;        //Water drainage in the second tank
  parameter Real r_1 = 5;        //Minimal required water level for
                                   the first tank
  parameter Real r_2 = 5;        //Minimal required water level for
                                   the second tank
  parameter Real u_1 = w - v_1; //Equation for x_1
  parameter Real u_2 = -v_2;     //Equation for x_2
  Real x_1(start = 10);          //Start value for the water in the
                                   first tank
  Real x_2(start = 10);          //Start value for the water in the
                                   second tank
equation
  der(x_1) = u_1;
  der(x_2) = u_2;
  //State change, therefore u_1 and u_2 need to be changed
  when x_2 <= r_2 then
  //The reinit function assigns the second argument to the first
      reinit(u_1, -v_1);
      reinit(u_2, w - v_2);
  end when;
  //State change, therefore u_1 and u_2 need to be changed
  when x_1 <= r_1 then
      reinit(u_1, w - v_1);
      reinit(u_2, -v_2);
  end when;
end WaterTank;
```

Figure 3.2: Water Tank in OpenModelica

Comparing it to the automaton from Fig. 2.3, there are similarities. For once, we see the same differential equations as the ones in the states q_2 and q_2. Further, the *when* cases in the equation part look exactly like the guard from the automaton. The only part that is missing is the domain of the state. Here, the HyAuLib helps.

So far, it looks like a reasonable model of the water tank. However, we should check the details and see what happens. We have the initial water levels of 10l in both tanks. Both tanks have a minimal required water level of 5l. Both holes have a

drainage of 1l. The water source can pour water with the constant rate of 1,8l into one tank at a time.

We start in the state q_1 of the automaton. The water source is above the first tank. In the equation part, we first state $der(x_1) = u_1$ and $der(x_2) = u_2$, while $u_1 = w - v_1$ and $u_2 = -v_2$. Here, we define the differential equation of x_1 and x_2. Thus, we modeled the state q_1 of the automaton, besides the fact that the domain is missing.

When the water level in the second tank drops to the minimal required water level or even below, we change the value of u_1 and u_2. By changing u_1 and u_2, the differential equations also change. After the change, the differential equations are $\dot{x_1} = -v_1$ and $\dot{x_2} = w - v_2$, which represents the state q_2 in the automaton. Now, the water level in the second tank rises again while the water level in the first tank drops. When the water level in the first tank drops to the minimal required water level or even below, the assignment for u_1 and u_2 change again to the initial assignment and the procedure starts all over again. We can see, even without the domains of the states, it seems that we have a precise model of the water tank system.

Let us now see what happens when we simulate the model of the water tank.

3.2.1 Simulation

At first, the simulation does exactly what we would expect, as Fig. 3.3 shows. First, the value of x_1 rises and x_2 drops. When x_2 drops down to 5, the transition is triggered and changes the state in the automaton and x_2 rises again.

Since there is less water getting poured into the tanks than its draining out, $w < v_1 + v_2$, the water levels are converging to the minimal water levels. However, the closer the variables x_1 and x_2 are to converging to the minimal required water level, a strange behavior sets in. Namely, the water in one tank drops monotonically, it would even drop below 0 and the water level of the other tank increases monotonically.

Not only the water tank system show such a behavior, but simulating the bouncing ball system from Fig. 3.1 shows a similar phenomenon in Fig. 3.4. While converging to 0, at some point the height of the bouncing ball becomes negative. Therefore, the bouncing ball would be under the surface on which it bounced. In the next chapter we will see how such a phenomenon can occur.

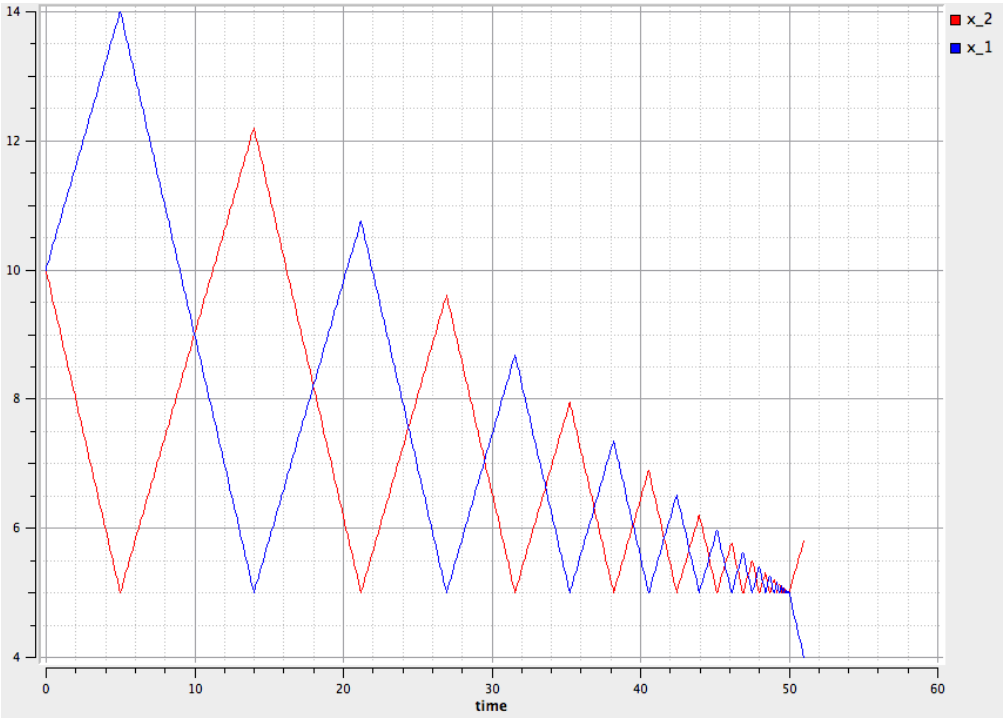

Figure 3.3: Water Tank Simulation in OpenModelica

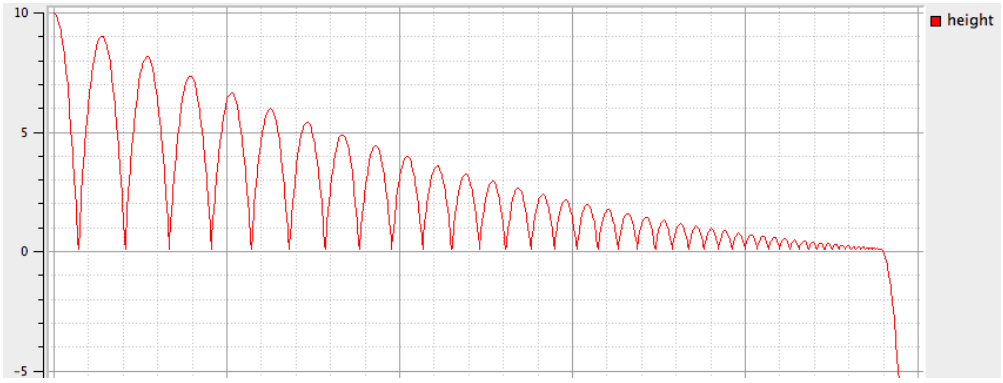

Figure 3.4: Bouncing Ball Simulation in OpenModelica

4 Zeno Behavior

Zeno behavior is a phenomenon that can only occur in models of physical systems, either hybrid [10] or temporal logic [17]. Therefore, such a behavior can not occur in a physical system, but only in the model of that system. Roughly speaking, a hybrid system shows zenoness in case it undergoes an unbounded number of discrete transitions in a finite and bounded length of time [10].

Zenoness is a problem introduced by abstraction, especially oversimplification [16],[11]. However, abstraction is needed to build a mathematical model of a system. Thus, we do need abstraction in order to validate hybrid systems. Besides the problem of abstraction, zeno behavior is also a problem of certain combination from the domain and the guard. Introducing zeno behavior is a risk we have to take due to the fact that validating hybrid systems is crucial and we need a model to validate the system.

Zeno behavior is not limited only to hybrid systems. Zenoness can also occur in systems based on temporal logic. Systems based on temporal logic normally have to adhere to timing constraints. A timed automaton models systems based on temporal logic. A way to deal with zeno behavior for timed automata is described in [17].

Throughout this chapter, we will see how such a phenomenon like zeno behavior can occur in a model of a hybrid system. We start by using the hybrid automaton of the water tank example to show how the model of the system can show zeno behavior. Afterwards, we give a definition when zeno behavior can occur. We end with the requirements for a hybrid automaton to be able to show zenoness.

4.1 Water Tank

Considering the hybrid automaton of the water tank system from Fig. 2.3, there are three relevant cases.

The first case $w > v_1 + v_2$ reflects the case that the water source adds more water to the tanks than water drains from the holes combined. In the simulation and the real world, the amount of water in the tanks rises constantly and therefore, no problem occurs.

In the case of $w = v_1 + v_2$, no problem occurs either. The water source adds the same amount of water into the tanks as water is draining from the holes combined. Therefore, the amount of water in the tanks stays the same. In Fig. 2.3, there are also guards to enter the automaton, namely $x_1 > r_1 \wedge x_2 > r_2$. Hence, one can enter the automaton only if there is at least as much water in each tank as required. Due to the fact that we have had enough water to begin with and that the amount of water in the tanks does not change, no problem can occur.

The third and last case is the most interesting. In the simulation of the water tank, we ran into trouble when we had the case of $w < v_1 + v_2$. Here, zeno behavior occurs. In the real world, both tanks would be empty at some point in time. In contrast, the model of the system yields that the water levels converge to the minimal required water level. To explain such behavior, we take a deeper look at the automaton of the system.

4.1.1 Zenoness

The domains and the guards of the automaton are crucial for understanding how the phenomenon of zenoness can occur. As a result from the domains and guards, we switch from q_1 to q_2 exactly when we reach $x_2 = r_2$ and the corresponding transition is triggered at $x_1 = r_1$.

Let us assume we start in state q_1, having the case of $w < v_1 + v_2$. Now the water level in the second tank decreases until it reaches its minimal required water level. At that point, the automaton can not operate in state q_1 anymore. The transition from q_1 to q_2 is triggered and the automaton switches to state q_2. Now, we operate in state q_2 until the water level in the first tank drops down to its minimal required water level. The automaton can not operate in state q_2 anymore, but switches to state q_1 again.

During every alternation, the overall water amount drops and the time needed to reach the minimal required water level decreases. The continuous variable x_1 is converging to r_1 and the continuous variable x_2 is converging to r_2. Reaching the point of convergence, $x_1 = r_1$ and $x_2 = r_1$, the automaton can not operate in either of the states anymore. If the automaton would operate in either state, the state would violate its domain. Thus, the automaton switches the states instantly after entering a state.

As we can see, the automaton undergoes an unbounded number of discrete transitions in a finite and bounded length of time. In case a hybrid automaton undergoes such behavior, it shows zeno behavior. Here, we also see the difference between the physical system and the hybrid automaton. In the physical system, the water

tanks will be empty at some point in time, while in the hybrid automaton, showing zenoness, the water level will never sink below the threshold.

The abstraction and the resulting model of the physical system are not completely reflecting the physical system. However, due to oversimplification, the model behaves the way we would want the physical system to behave. One problem we can see here, is that the abstraction can introduce faults. In our case, it does not take into consideration the time the physical system needs to switch the water source from one tank to the other. While the physical system needs some time to change from one water tank to the other, the hybrid automaton switches from one state to another instantly.

4.1.2 Simulation

In Fig. 3.3 and Fig. 3.4, we see a rather strange behavior, while the continuous variables, x_1 and x_2, converge to the minimal required water level. What we see here, can be explained by zeno behavior paired with numerical integration methods. Simulations of hybrid systems use numerical integration for the computation of the differential equations. Even using different numerical integration methods can result in different outcomes for the same simulation [3].

Due to the fact that we have infinite state transitions in a finite time, in case the system undergoes zeno behavior, the behavior we see in Fig. 3.3 and Fig. 3.4 can be explained. For every state transition, an error correction has to take place using numerical integration. Having infinitely many transitions, these error correction may not work correctly anymore and such a behavior can occur [3].

4.2 Zeno Set

For determining the zeno set, we need to examine the zeno executions. Zeno executions are runs of the system that show zeno behavior. A point $(q, x) \in Q \times X$ is a zeno point if it is a convergence point of a zeno execution with infinite transitions in a finite time. During simulating, if the continuous variables converge to a zeno point the simulation might yield a false result. The zeno set of a zeno execution is the set of all its zeno points. Having all of the zeno points in a set, makes it easier to validate our simulation results by checking that no continuous variable is converging to a zeno point. In this case, the result is not falsified. If a continuous variable converges to a point in the zeno set, the simulation can show zenoness. Then, the modeler has to check again whether the zenoness falsified the outcome of the simulation. $Z_\infty \subset Q \times X$ denotes the zeno set [20].

There are also different zeno types. In case all executions show zeno behavior, the hybrid system is called strongly zeno. A hybrid system is called (weakly) zeno if at least one run in the system is a zeno run [10].

4.3 Reset Map

The reset map is called *identity* if the value of a continuous variable does not change during a transition. A reset map is called *non-expanding* in case:

$$\exists \delta \in [0,1] s.t. \forall e \in E, x \in G(e), x' \in R(e,x) : \\ ||x'|| \leq \delta ||x||.$$

$||x||$ denotes the euclidean norm of x. The *non-expanding* reset map yields that the new values of the continuous variables after a state transition can either stay the same as described in the *identity* case or decrease. The reset map could be used to model a damping coefficient or to simulate a delay.

4.4 Detecting a Zeno Set

Detecting a zeno set can be achieved by applying certain steps, which we are going to explore in this section. First of all, we have to determine when a hybrid automaton can show zeno behavior to make a statement about the zeno set. A cycle is a prerequisite for a hybrid automata to accept zeno executions, which is formulated in Proposition 1. $Reach_H$ denotes the set, which contains all reachable states for the automaton H.

Proposition 1. *If there exists a finite collection of states* $\{(q_i, x_i)\}_{i=1}^{K}$ *such that*

- $(q_1, x_1) = (q_K, x_K)$
- $x_{i+1} = (q_i, q_{i+1}, x_i), \forall i = 1, ..., K-1$ *(After taking a transition, the values of the continuous variables are defined by the reset map); and*
- $(q_i, x_i) \in Reach_H, \exists\, i = 1, ..., K$*;*

then the hybrid automaton accepts a zeno execution [20].

Using the reset map, it is possible to make statements about the zeno set in case the automaton accepts a zeno execution. Before we can give a statement about the zeno set, we need a few definitions. Let $\bar{U}$ denote the closure of the set U, U^0 its interior, and $\partial U = \bar{U} \setminus U^0$ its boundary. Further, we call the set of states, which are infinitely often visited, Q_∞.

Theorem 1. *A hybrid automaton with a* non-expanding *reset map, which accepts zeno executions, has the following zeno set:*
$Z_\infty = \{(q_i, x_i)\}_{i=1}^m, m > 0$
If $G(q, q') \cap D(q)^0 = \emptyset, \forall (q, q') \in E$ *with* $q, q' \in Q_\infty$
then $x_i \in \partial D(q_i)$ *for all* $i = 1, ..., m$.

Thus, the zeno set of a hybrid automaton with a *non-expanding* reset map accepting zeno executions consists of the boundaries of the domain if the intersection of the guard and the interior of the domain is empty for every transition of the cycle.

Using this result, the following non-zeno condition follows.

Corollary 1. *A hybrid automaton with* identity *reset map does not accept zeno executions if*

- $G(q, q') \cap D(q)^0 = \emptyset, \forall (q, q') \in E$,
- *for all cycles* $\{q_i\}_{i=1}^K$ *with* $q_K = q_1$ *and* $(q_i, q_{i+1}) \in E$,

  $1 \leq i \leq K-1, \cap_{i=1}^{K-1} \partial D(q_i) = \emptyset$.

Hence, a hybrid automaton does not accept zeno executions if for every transition of the cycle the intersection of the guard and the interior of the domain is empty, and if for every cycle, the intersection of the boundary of the domains is empty.

Theorem 1 and Corollary 1 correspond to Theorem 3 and Corollary 1 from [20].

Cycle	I	NE	$G(q, q') \cap D(q)^0 = \emptyset$	Zeno Set	Zeno Behavior
$\times$	$-$	$\times$	$\times$	$x_i \in \partial D(q_i)$	YES
$\times$	$-$	$\times$	$-$	No clear statement	MAYBE
$\times$	$\times$	$-$	$-$	No clear statement	MAYBE
$\times$	$\times$	$-$	$\times$	$\cap_{i=1}^{K-1} \partial D(q_i) = \emptyset$	NO
$\times$	$\times$	$-$	$\times$	$\cap_{i=1}^{K-1} \partial D(q_i)$	YES
-	do not care			$\emptyset$	NO

Table 4.1: Conditions for Zeno Behavior

Table 4.1 contains all possibilities whether a hybrid automaton can show zeno behavior. In every case but the last, the automaton contains a cycle. The first case

reflects Theorem 1. The hybrid automaton possesses a *non-expanding* (NE) reset map. Thus, the automaton allows zeno executions. Furthermore, the intersection of the guard and the interior of the domain is empty for every transition of the cycle. Hence, the boundaries of the domain are elements of the zeno set. Since the zeno set is not empty, it is possible for the automaton to show zeno behavior.

The second case has nearly the same setup as the first. The only difference is that the intersection, for every transition of the cycle, of the guard and the interior of the domain is not empty. Therefore, we have an overlapping interval between the guard and the domain. Such an overlapping interval makes it impossible to determine when the automaton triggers the transition. Hybrid automata are nondeterministic, hence, the transition can be triggered at any time after the guard holds true. Having an overlapping interval, it is also possible for the automaton to keep operating in the current state. Hence, zeno behavior could occur if the transition would always trigger instantly, but does not if the automaton operates longer in the state. The same argumentation holds true for the third case. In both setups, we can not make a clear statement about the zeno set. Even more, zeno behavior can occur in the automaton but not necessarily, resulting in *MAYBE* zeno behavior.

The fourth and fifth case relates to Corollary 1. The automaton possesses an *identity* (I) reset map. Furthermore, the intersection of the guard and the interior of the domain is empty. Now, the intersection of the boundaries of the domain for every state in the cycle is important. In the fourth case, the intersection is empty. Therefore, the automaton can not show zeno behavior. In the fifth case, the intersection is not empty. Due to Corollary 1, the automaton is not zeno free but shows zeno behavior. The zeno set consists of the intersection of the boundaries of the domain for every state in the cycle.

Finally, we have the case of no cycle. Having no cycle implies that the automaton can not show zeno behavior. Theorem 1 and Corollary 1 and therefore, Table 4.1 are used for the zeno detection in the tool.

5 Tool for Zeno Set Detection

Knowing what zeno behavior is and when it can occur, we now focus on detecting it automatically. In this chapter, we will take a closer look at the tool for detecting zeno sets which I implemented. The tool for zeno set detection is a command-line tool written in Haskell. Generally, the tool goes through the following steps: The tool takes an automaton in a file as input. It parses the automaton into a data structure. The tool uses the algorithm of Robert Tarjan, "Enumeration of the Elementary Circuits of a Directed Graph" [19], to detect cycles. The cases described in Table 4.1 form the basis for the implementation of the following part of the tool. After the cycle detection, the tool checks whether the automaton contains any cycles. If the tool finds a cycle, it continues with the calculation of the reset map of the first cycle. Otherwise, it stops and generates the output that no zeno behavior can occur.

The next step is to intersect the guard with the interior of the domain for every transition in the cycle. After the computation of the intersection, the tool checks if all intersections are empty. In case the intersections are not empty, the tool proceeds to generating the output for the overlaps in a readable format. Otherwise, the tool has to consider the reset map. For the *non-expanding* reset map, the tool calculates the zeno set as described by Theorem 1. By calculating the boundaries of the domain for every state, the tool generates the zeno set for the cycle and stores the result. For the *identity* reset map, the tool first has to check whether the intersection of the boundaries of the domains of every state in the cycle is empty. In case the intersection of the boundaries is empty, we know that the cycle is zeno-free. Otherwise, the intersection is the zeno set, which is saved after being formatted in a readable output. Thereby, the tool covers all cases of Table 4.1. In case more than one cycle has been detected, the tool starts over with the next cycle until all cycles are evaluated. At the end, the tool prints the results of the evaluation of all cycles, i.e., the overlap, the zeno set, or the observation that no zeno behavior can occur. To picture the steps of the program more clearly, a control flow graph is given in Fig. 5.1 in the next subsection.

The tool was developed using Mac OS X, version 10.7.4, with GHCi, version 7.0.4. To execute the tool, the library Data.IntervalMap.Interval is needed. It can be downloaded using cabal (cabal install IntervalMap). Version 0.2.3.3 of Data.IntervalMap.Interval was used for the development.

5.0.1 Control Flow Graph

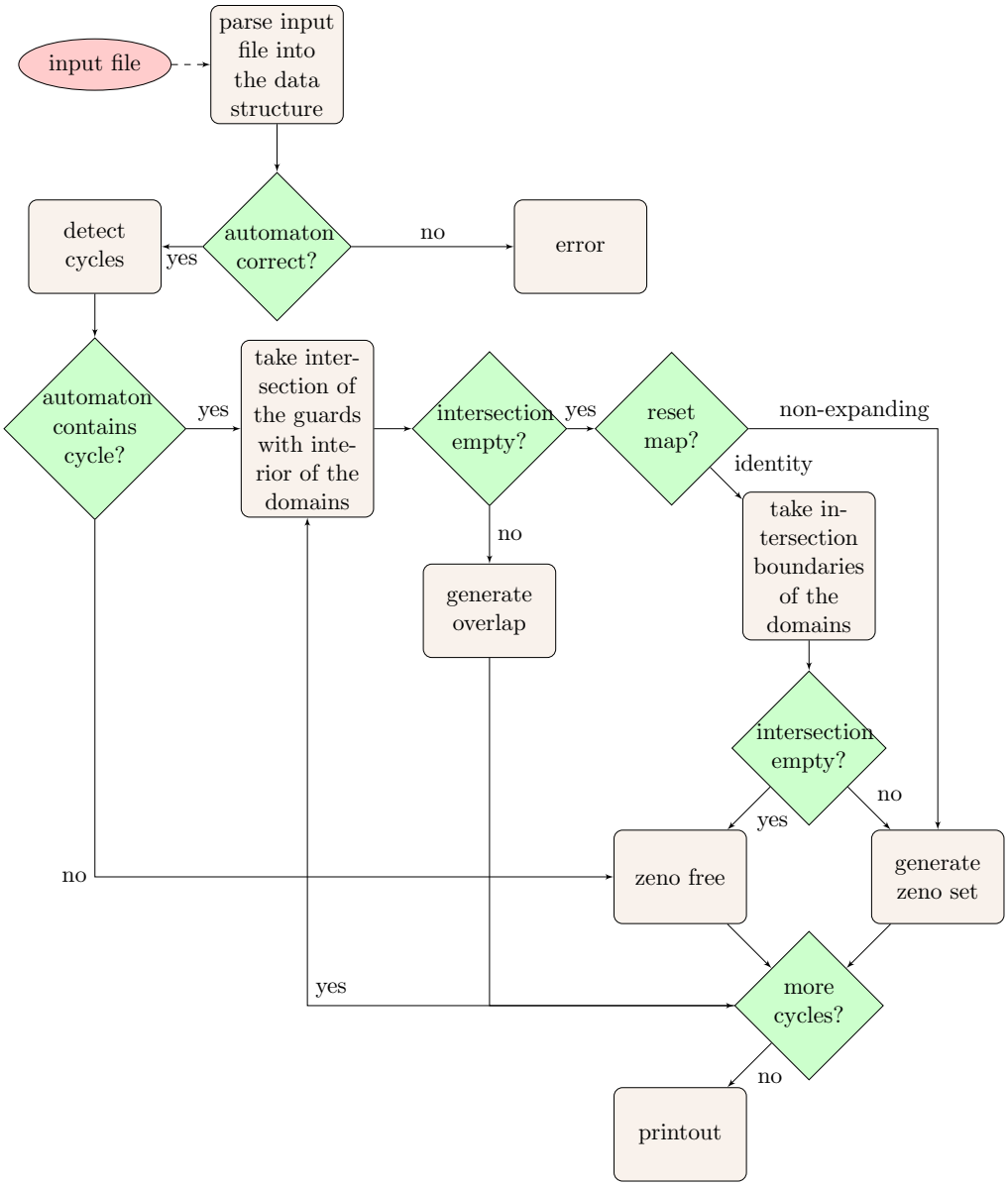

Figure 5.1: Control Flow Graph of the Tool

In the next section, we give requirements the modeler has to follow to ensure that the tool works correctly.

5.1 Specification

There are a few properties of the automaton in the input file that have to hold. First, we have the grammar which defines the syntactic structure of the input files. Second, we have general properties. The general properties add requirements not captured by the grammar as well as some invariants.

5.1.1 Automaton Grammar

A file has to contain exactly one automaton. The grammar of the file is defined the following way:

$$\text{“}Automaton;\text{”}\,((State|Transition)\text{“};\text{”})^+$$

$$\begin{array}{ll}
State & = \text{“}State,\text{”}\ Name\ \text{“},\text{”}\ Start\ \text{“},\text{”}\ Domain\ \text{“},\text{”}\ Equations \\
Name & = [a-z, A-Z, 0-9]^+ \\
Start & = True\ |False \\
Domain & = Conditions \\
Equations & = Equation|Equation(\text{“},\text{”}\ Equation)^+ \\
Equation & = [a-z, A-Z, 0-9, =, \text{␣}]^+ \\
 & \\
Transition & = \text{“}Transition,\text{”}\ Source\ \text{“},\text{”}\ Dest\ \text{“},\text{”}\ Guard\ \text{“},\text{”}\ Reset \\
Source & = Name \\
Dest & = Name \\
Guard & = Conditions \\
Reset & = 1|(0.[0-9]^+) \\
 & \\
Conditions & = Condition(\text{“␣}AND\text{␣”}\ Condition)^* \\
Condition & = ContinuousVariable\ \text{“␣”}\ Comp\ \text{“␣”}\ (Var|Number) \\
ContinuousVariable & = ([0-9]^*[a-z, A-Z]^+[0-9]^*)^+ \\
Var & = ([0-9]^*[a-z, A-Z]^+[0-9]^*)^+ \\
Number & = ([0-9]^+)|([0-9]^+).([0-9]^+) \\
Comp & = (<\ |\ >)|((<\ |\ >\ |\ =)\ =)
\end{array}$$

5.1.2 General Properties

The additional properties that have to hold are:

- The hybrid automaton has to be correct and specified as a set of states and transitions.

- Neither the state set nor the transition set is permitted to be empty.
- There exists only one transition from one state to another. Therefore, the combination of source and destination is unique in the set of transitions. More formally said, the transitions have to be deterministic.
- A continuous variable must not be restricted more than twice in either a guard or a domain.
- In case a continuous variable is bounded with another continuous variable, the reversed case has to be inserted, meaning $x_1 <= x_2$ and $x_2 >= x_1$.
- The boundaries of the guard of a transition have to lie inside the boundaries of the destination state in case the continuous variables are bounded from the same side in both state and guard.
- If a continuous variable is restricted twice, it must be restricted with $< \mid <=$ and $> \mid >=$, but not any other combination.

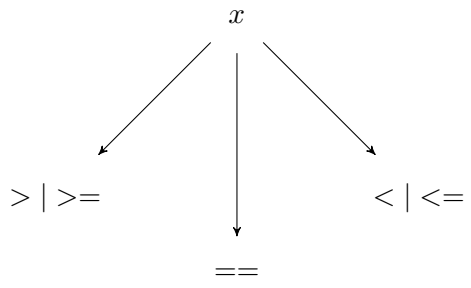

Figure 5.2: Boundaries of a Continuous Variable of the Tool

To elaborate on the last point, Fig. 5.2 shows the operations that can be applied to a continuous variable. If only one restriction is applied on a continuous variable, no problem can occur. For the case of having two restrictions, one from above and one from below is the only reasonable pair. However, if a continuous variable is bounded twice from the same side in either a guard or a domain, the tool does not consider the second restriction. Having more than two restriction is not reasonable either.

If the automaton of the input file violates either of these properties, I can not guarantee a correct result of the tool.

5.2 Tool Description

In this section, we give a more detailed description of the tool. Further, we will see how the tool makes use of the requirements defined in the previous section.

5.2.1 Parsing and Cycle Detection

The tool requires a representation of a hybrid automaton in Haskell. A data structure for a hybrid automaton is used for that purpose. The data structure allows parsing of the automaton from the input file. While parsing the input file, the tool parses every line separately. The end of a line in the file is represented by the symbol ";". Due to the grammar, the first line of the file must only contain the word "Automaton". If the first line does not start with "Automaton", the execution stops and returns an error. Otherwise, the tool proceeds to parse the next line. From now on, every new line has to start with either the word "State" or "Transition", as defined by the grammar. In case the new line does not start with one of the words, the tool returns an error.

If the new line starts with the word "State", the tool checks whether all elements of the state are defined. By separating the line after each "," the tool retrieves the elements and checks whether the line consists of at least four elements after "State" indication. Otherwise, it returns an error. Next, the tool proceeds to parse every element by itself. While parsing the elements the order, as specified in the grammar, is crucial, e.g. the first element after "State" is parsed as the name of the state. For the conditions, we recursively check if there is an "AND", and combine the restrictions. In case there are more than four elements, the state has more than one equation. These elements are recursively added to the other equations. If either of the elements do not meet the grammar, the tool returns an error.

In case the new line starts with the word "Transition", the tool behaves similarly to the case of "State". It checks whether there are at least four elements separated by the symbol "," after the "Transition". If there are less than four elements, the tool returns an error. In case enough elements can be found, the elements are read as follows: The first element is parsed as the source of the transition. The second element is translated into the destination. The source as well as the destination should refer to the name of a state. Parsing the third element, the conditions, works exactly like parsing the condition in a state. The tool recursively parses the input file until all lines are parsed.

After the file is parsed, the tool continues with a cycle detection. For the cycle detection, I implemented the algorithm "Enumeration of the Elementary Circuits

of a Directed Graph" [19] of Robert Tarjan. Due to the fact that an automaton has directed edges, the automaton can also be represented as a directed graph and therefore, the algorithm can be applied. Unfortunately, the algorithm assumes that the graph has no reflexive edges. As the algorithm forms the basis for the cycle detection, reflexive edges might not always be handled correctly. The tool generates an adjacency matrix out of the transitions. The adjacency matrix is crucial to run the algorithm of Robert Tarjan. With the adjacency matrix as input to the algorithm, the tool can retrieve all cycles of the automaton.

5.2.2 Zeno Detection

Let us now go into detail about how the zeno detection works. Having the cycles, the implementation of the cases of Table 4.1 can start. At the beginning, the tool checks whether the automaton contains any cycles. In case the automaton consists of no cycles, the execution stops and tells the user that the automaton does not show zeno behavior as it contains no cycles. Otherwise, the tool proceeds with the zeno detection. The first step is to retrieve the reset map for the cycle. The reset map is calculated by summing up all reset maps in the transition and dividing the sum by the length of the cycle. In case the result is 1, we have an *identity* reset map, otherwise a *non-expanding* reset map, since the reset map , due to the grammar, should be ≤ 1. However, at first, both branches continue along the same way.

General Idea

The tool checks the intersections of the interior of the domain with the guard for all transitions in the cycle if they are empty. More formally said, $G(q, q') \cap D(q)^0 = \emptyset, \forall (q, q') \in E$ with $q, q' \in Q_\infty$. To check the intersection, the tool has to retrieve the domain and the guard. For the guard, the tool only has to take the transition into account. However, for the domain, it is not that simple. The domain does not only consist of the restrictions in the state, but also of the guard from the incoming transition in the cycle. To get a better understanding, consider the water tank example in Fig 2.3. The domain of q_1 should not only contain the restrictions for q_1, but should also be bounded by the incoming transition. The domain should contain $x_2 \geq r_2$ and $x_1 \leq r_1$. Considering the cycle of q_1 and q_2, it is obvious. The automaton can only operate in the state as long as $x_2 \geq r_2$ holds true. In addition, the state q_1 can only be entered when $x_1 \leq r_1$ holds true. Hence, the domain does not only define how long the automaton can operate in the state, but also when it can start to operate in the state.

The intersection is implemented in the following way: The tool starts with retrieving the names of the first two states in the cycle. The first state represents the source, or q, while the second state is the destination, or q'. For the domain of q, the tool retrieves the restrictions of state q as well as the guard of the incoming transition to q, which lies in the cycle. Next, the guard of the transition from state q to q' is retrieved.

The tool intersects every continuous variable separately. For the intersection of a continuous variable, the tool looks at the first restriction of the guard. The tool selects the continuous variable that is restricted in the differential equation of the first guard. Next, the tool checks if that continuous variable is bounded in any other restriction of the guard. Having all restrictions of the guard, which bound that continuous variable, the tool generates the interval spanned by the restrictions. Afterwards, the tool checks the domain. The tool recursively searches through all restrictions on the domain and retrieves the restrictions on the continuous variable. Those restrictions are used to generate the interior of the domain for that continuous variable. When generating the interior of the domain, the restrictions of the state are weighted more than the ones from the incoming guard. State restrictions are weighted more, since the guard of an incoming transition must lie inside the boundaries of the state. After the guard and the interior of the domain are calculated, the tool takes the intersection of them. With the last step, the computation of the intersection of the guard with the interior of the domain is completed for one continuous variable. Now, the procedure starts all over again and looks for the next continuous variable that is restricted by the guard. In the following subsections, we will see how the general idea is implemented.

Implementation of Constant as Bound

Table 5.1 shows how the interval of the guard is constructed. Here, the tool uses the property that a continuous variable is bounded at most twice in a guard. Since the property of reasonable bounds holds, all cases are covered in Table 5.1. The guard is constructed using the *Data.IntervalMap.Interval* library as pictured in Table 5.1. The *Data.IntervalMap.Interval* library gets the lower and upper bound and generates the interval for a continuous variable in the guard.

For the case of calculating the interior of the domain, we do not only have to account for the restrictions made in the state, but also for the guard of the incoming transition in the cycle. Therefore, we know that we have at most four restrictions, since the state as well as the guard can restrict a continuous variable at most twice. While calculating the intervals, the assumption is important that the boundaries of the guard lie inside the boundaries of the domain.

Guard			
Lower	Upper	Equal	Interval
×	×	–	$[l..u]$
×	–	–	$[l..\infty]$
–	×	–	$[-\infty..u]$
–	–	–	$[-\infty..\infty]$
–	–	×	$[e]$

Table 5.1: Retrieving the Interval of a Guard for a Continuous Variable

Table 5.2 shows how the property is used to determine the interior of the domain. Besides that property, the implementation makes use of the fact that the restrictions of the state are retrieved before the restrictions of the guard. Therefore, the first restrictions are the stronger ones. For example, if we bound a continuous variable in the domain twice from above, the tool uses the first retrieved restriction, since that is the one of the state. In fact, the tool just works on two restrictions at once, since two restrictions are enough to model the lower and upper bound. Hence, if there are two bounds from the same side and there are still restrictions in the domain, the function calls itself again without the second bound from the same side. Thus, we achieve to implement all cases from Table 5.2. The boundaries of the domain are constructed using the *Data.IntervalMap.Interval* library, by passing it the lower and upper bound.

Having the interior of the domain and the guard, the tool can take the intersection. The Data.IntervalMap.Interval library provides an overlap function. The overlap function checks whether two intervals overlap. In case the intervals overlaps, the intersection is not empty and we print out the overlap. Unfortunately, the *Data.IntervalMap.Interval* library does not provide a function to determine the interval of the overlap of two intervals. For that purpose, I implemented a function that returns the interval of the overlap of two intervals. The function is called when the overlap function returns that the intersection actually has an overlap. If the intervals do not overlap, the tool returns the empty set.

Implementation of Variable as Bound

For the case that the continuous variable is bounded by a variable, the tool takes the intersection as follows: The tool checks the relational operator and the variable. The tool goes through the restrictions of the domain and tries to match the variables of the restrictions with the variables from the guard. As declared before, a state can have at most four restrictions on the domain.

Domain						
State Restriction			Incoming Guard			
Lower1	Upper1	Equal1	Lower2	Upper2	Equal2	Interval
×	×	–	do not care			$(l_1..u_1)$
×	–	–	×	–	–	$(l_1..\infty)$
×	–	–	×	×	–	$(l_1..u_2)$
×	–	–	–	×	–	$(l_1..u_2)$
×	–	–	–	–	–	$(l_1..\infty)$
×	–	–	–	–	×	$(l_1..\infty)$
–	×	–	×	–	–	$(l_2..u_1)$
–	×	–	×	×	–	$(l_2..u_1)$
–	×	–	–	×	–	$(-\infty..u_1)$
–	×	–	–	–	–	$(-\infty..u_1)$
–	×	–	–	–	×	$(-\infty..u_1)$
–	–	–	×	–	–	$(l_2..\infty)$
–	–	–	×	×	–	$(l_2..u_2)$
–	–	–	–	×	–	$(-\infty..u_2)$
–	–	×	–	–	–	$[]$
–	–	×	–	–	×	$[]$
–	–	–	–	–	×	$[]$
–	–	–	–	–	–	$(-\infty..\infty)$

Table 5.2: Retrieving the Interval of the Interior of a Domain for a Continuous Variable

Having only one restriction on the guard, the tool tries to match the variable of the guard with the ones of the domain. In case a variable from the domain matches the variable of the guard, the tool proceeds and takes the intersection. The first match can simply be taken, since the state restrictions are retrieved first as they are weighted more. Table 5.2 shows the possible restrictions on a continuous variable and demonstrates that the state restrictions are weighted more.

If a continuous variable is bounded twice in the guard, the tool checks the restrictions on the domain recursively to match both of them. While trying to match them, it is possible that only one variable can be matched. Having found at least one match, the tools proceeds by comparing the operator whether an overlap exists.

The operator comparison checks if the variables are bounded by the same category of operators. < and <= as well as > and >= belong to the same kind of operator for our purpose. For the operators of one category, no difference for the interior and closure exists. In case a variable is bounded by the same category in both domain and guard, an overlap exists. Every other combination of operators results in no overlap. If no overlap exists, the tool returns an empty set, otherwise the overlap.

After the tool calculated the intersection for one continuous variable, it proceeds with the next one from the guard. When all restrictions on the guard are evaluated, the tool checks if the domain holds continuous variables, which are not restricted in the guard. For those, the tool generates the overlap. Having calculated the intersection for every continuous variable, the tool checks if the empty set was returned at least once. If the empty set is returned, the vector of all continuous variables does not intersect in at least one dimension. Therefore, the interior of the domain and the guard do not intersect at all.

Afterwards, the tool moves forward in the cycle. The tool recursively takes the intersection for the next state and transition until the tool evaluated the whole cycle. In the recursion, the state that previously was the destination becomes the new source and the next state becomes the destination. Having evaluated every transition in the cycle, the tool evaluates the intersections. In case the intersection is empty, the tool can proceed to calculate the zeno set. The intersection is empty if every intersection of the interior of the domain with the guard is empty. Otherwise, the tool generates the output for the overlap, since we can not make any clear statement about zenoness.

5.2.3 Determining the Zeno Set

This subsection applies if the intersection calculated in the previous subsection is empty. For determining the zeno set, the calculated reset map is of importance. If the reset map is *non-expanding*, the tool retrieves the boundaries for each state and stores them as the result. Retrieving the boundaries of the domain follows the same procedure as retrieving the interior. First of all, the tool retrieves the restrictions for the domain the way described in the Subsection 5.2.2 “Zeno Detection”. While generating the domain, the same properties have to hold true as before when the domain for the intersection was generated. The boundaries of the domain are constructed separately for each continuous variable.

To construct the boundaries of a continuous variable, the tool behaves similarly to the part of the intersection with a variable as bound. The tool tries to find bounds from both sides. However, in case a continuous variable is bounded twice from one side, the restriction of the state get weighted more as shown in Table 5.3. Table 5.3 depicts the boundaries that follow from varying restrictions.

Fig. 5.3 shows how the boundaries of a continuous variable are retrieved in the tool. Var1 indicates a constant bound of the restriction. Var2 denotes a variable. In case the continuous variable is only restricted once, retrieving the boundaries is straightforward. The variable or constant that is restricting the continuous variable

Domain						
State Restriction			Incoming Guard			
Lower1	Upper1	Equal1	Lower2	Upper2	Equal2	Interval
×	×	–	do not care		–	$[l_1, u_1]$
×	–	–	×	–	–	$[l_1]$
×	–	–	×	×	–	$[l_1, u_2]$
×	–	–	–	×	–	$[l_1, u_2]$
×	–	–	–	–	–	$[l_1]$
×	–	–	–	–	×	$[l_1]$
–	×	–	×	–	–	$[l_2, u_1]$
–	×	–	×	×	–	$[l_2, u_1]$
–	×	–	–	×	–	$[u_1]$
–	×	–	–	–	–	$[u_1]$
–	×	–	–	–	×	$[u_1]$
–	–	–	×	–	–	$[l_2]$
–	–	–	×	×	–	$[l_2, u_2]$
–	–	–	–	×	–	$[u_2]$
–	–	×	–	–	–	$[e_1]$
–	–	×	–	–	×	$[e_1]$
–	–	–	–	–	×	$[e_2]$
–	–	–	–	–	–	[]

Table 5.3: Generating the Boundaries of a Continuous Variable in a Domain

is saved as result. The only difference in the implementation is that the tool has to convert a constant to a string, while a variable is defined as a string already.

The first two cases represent the fact that the continuous variable is bounded from above and below. Cases three to four adhere to the == as operator. In case both restrictions bound the continuous variable with the == operator to one point which is not the same point, the tool returns an error since that is not possible. Else, if the continuous variable is bounded to the same point, that point is returned. Another case, which represents the fourth case, is having the equal operator only in the guard of the incoming transition. Due to the assumption of reasonable multiple restrictions, the tool can assume that the second restriction has to be in the guard. Furthermore, using the assumption that the bounds of the guard lie inside the restrictions of the state, the tool does not have to consider the restriction with the equal operator. Additionally, there should not be any restrictions following due to the assumption of only reasonable restrictions. Hence, the tool checks if there are no more restrictions to consider. In that case, the tool returns the boundary.

```
> makeBoundaries :: [Condition] -> Name -> [String]
> makeBoundaries [(Cond _ a b)] _ = (getVarName b) : []

> makeBoundaries ((Cond e a (Var2 b)):(Cond _ c (Var2 d)):xss) z
>       | ((a == "<" || a == "<=") && (c == ">" || c == ">=")) =  if b == d then b : [] else d : b : []
>       | ((c == "<" || c == "<=") && (a == ">" || a == ">=")) =  if b == d then b : [] else b : d : []
>       | a == "==" && c == "==" = if b == d then  b : [] else error (show b ++ " and " ++ show d ++ " are both restricting the variable " ++ show z ++ " with ==")
>       | c == "==" = if null xss then b : [] else error ("Reasonable restriction assumption in state " ++  (show z) ++ " violated.")
>       | ((a == "<" || a == "<=") && (c == "<" || c == "<=")) = if null xss then b : [] else makeBoundaries ((Cond e a (Var2 b)):xss) z
>       | ((a == ">" || a == ">=") && (c == ">" || c == ">=")) = if null xss then b : [] else makeBoundaries ((Cond e a (Var2 b)):xss) z
>       | otherwise = error ("Could not determine which variable, should restrict " ++ show z ++ " either " ++ show b ++ " or " ++ show d)

> makeBoundaries ((Cond f a (Var1 b)):(Cond _ c (Var1 d)):xss) z
>       | ((a == "<" || a == "<=") && (c == ">" || c == ">=")) = if b == d then (show b) : [] else (show d) : (show b ): []
>       | ((c == "<" || c == "<=") && (a == ">" || a == ">=")) = if b == d then (show b) : [] else (show b) : (show d): []
>       | a == "==" && c == "==" = if b == d then  (show b) : [] else error (show b ++ " and " ++ show d ++ " are both restricting the variable " ++ show z ++ " with ==")
>       | c == "==" = if null xss then (show b) : [] else error ("Reasonable restriction assumption in state " ++  (show z) ++ " violated.")
>       | (a == "<" || a == "<=") && (c == "<" || c == "<=") = if null xss then (show b) : [] else makeBoundaries ((Cond f a (Var1 b)):xss) z
>       | (a == ">" || a == ">=") && (c == ">" || c == ">=") = if null xss then (show b) : [] else makeBoundaries ((Cond f a (Var1 b)):xss) z
>       | otherwise = error ("Could not determine which variable, should restrict " ++ show z ++ " either " ++ show b ++ " or " ++ show d)
```

Figure 5.3: Function to Retrieve the Boundaries of a Continuous Variable

Otherwise, an error is returned. The last cases handle two bounds from the same side. Restrictions from the state are weighted more. Therefore, the function calls itself recursively without the second restriction if there are more restrictions to be considered in *xss*. Otherwise, the boundaries are constructed using the first restriction. Following that procedure, the tool behaves as pictured in Table 5.3.

After generating all boundaries of the domains in the described way, the tool can proceed. In case the reset map is *non-expanding*, the tool stores the zeno set. The zeno set consists of the state combined with the boundaries of that domain. Otherwise, having an *identity* reset map, the tool has to check if the automaton can show zeno behavior. If the intersection of the boundaries of the domains for each state in the cycle is empty, then the cycle is zeno free. We take a deeper look at how the intersection of the boundaries of the domains works in the next subsection.

5.2.4 Boundary of the Domain Intersection

For the boundary intersection, the tool goes through the cycle and takes, for every state and its successor in the cycle, the intersection of the boundaries of the domains. Intersecting the boundaries is also similar to the intersection of the interior of the domain and guard. It is done for each continuous variable separately. Fig. 5.4 shows how the intersection of a continuous variable is implemented.

```
1 > intersectTwoStates ::[(Name, [Name])] -> [(Name, [Name])] ->
      [(Name, [Name])]
2 > intersectTwoStates [] _ = []
3 > intersectTwoStates _ [] = []
4 > intersectTwoStates ((a,b):xs) ((c,d):ys)
5 >                    | a == c = (a,intersect d b) :
      intersectTwoStates xs ys
6 >                    | a > c = intersectTwoStates ((a,b):xs) ys
7 >                    | otherwise = intersectTwoStates xs ((c,d):ys)
```

Figure 5.4: Function for the Intersection of the Boundaries of a Continuous Variable

The function *intersectTwoStates* has two lists of tuples as input. Both lists represent a state in the cycle. The second list reflects the successor state of the first list. The tuple (Name, [Name]) consists of the continuous variable in the first argument and the boundaries for that continuous variable as a list in the second argument. Further, the continuous variables are ordered ascending. Using the fact that the continuous variables are ordered, the tool goes through both lists and check if the heads of the

lists are equal. Otherwise, the tool drops the smaller head and calls the function recursively.

If the continuous variables match, the tool checks whether there is an intersection of the boundaries from the continuous variable. In Haskell, there is an intersection function implemented. The function takes two lists as input and returns a list of all elements, that are in both lists. Therefore, if the boundaries have an empty intersection, the intersection function returns the empty list. Otherwise, the function returns the elements of the intersection. The tool concatenates a tuple of the continuous variable and the result of the intersection of the two lists to the recursive call of the *intersectTwoStates* without the heads of the lists. When either of the lists is empty the function terminates and returns either an empty list, which represents the intersection free case, or the tuple of continuous variable and intersection.

For the intersection of the boundaries, it is sufficient if in one continuous variable or dimension, no intersection exists. Then the intersection of the two domains is empty. Otherwise, if the intersection for every continuous variable is not empty, the intersection of the two domains is not empty as well. After having calculated the intersection for the boundaries of each domain in the cycle, the tool generates and stores the output for that cycle. The output consists of the evaluated cycle and its zeno behavior, followed by the zeno set. If the automaton consists of more than one cycle, the tool starts again with the intersection of the domains and guards, until all cycles are evaluated. After all cycles have been evaluated the tool prints out all generated and stored outputs, as shown in Fig. 5.1.

6 Evaluation

Knowing how the tool works, we still have to verify that the implementation is correct. Therefore, we test the tool against a known result with the examples that accompanied us throughout the thesis in Section 6.1 - 6.3. We check the example of the air conditioning system, which does not show zeno behavior, in Section 6.1. In Section 6.2, we test that the tool detects a cycle with a *non-expanding* reset map which shows zeno behavior, by testing it against the bouncing ball system. The water tank example is the last example left. In the last example, we check the tool against an automaton with a cycle containing an *identity* reset map, which shows zeno behavior, in Section 6.3. In Section 6.4, we will have a look at an example of an automaton with overlapping interval. To conclude this chapter, we conduct a completeness evaluation in Section 6.5.

6.1 Air Conditioning

First of all, we check the air conditioning example from the hybrid systems section. Fig. 2.1 shows the automaton that we have to translate into our grammar. The translated input file is shown in Fig. 6.1.

```
1 Automaton;
2 State, on, True, t <= 20, t = 1;
3 State, off, True, t >= 15, t = -1;
4 Transition, on, off, t >= 20, 1;
5 Transition, off, on, t <= 15, 1;
```

Figure 6.1: Input File for the Air Conditioning System

For the air conditioning system, a cycle is found by the tool. printCycle is a test function that returns in a list every cycle of the automaton as a list.

**Zeno> printCycle "air.txt"*
[["on","off","on"]]

In case we would define the transitions in the reversed order, the cycle would be *[["off","on","off"]]*. As we can see, the cycle detection algorithm works for the air conditioning system. The cycle *[["on","off","on"]]* possesses an *identity* reset map. Hence, Corollary 1 is of interest for this system. First, we check whether the first part of the assumption, $G(q,q') \cap D(q)^0 = \emptyset, \forall (q,q') \in E$, holds. The guard of the transition from the *on* state to the *off* state is $t >= 20$. Therefore, the corresponding interval of the continuous variable t is $[20..\infty)$. Computing the interior of the domain of the *on* state, we have to consider the guard from the incoming transition as well as the restrictions of the *on* state. The restrictions $t <= 20$ from the state and $t <= 15$ from the incoming transition are used to generate the interval. The resulting interval is $(-\infty..20)$ for the continuous variable t in the interior of the domain of the *on* state. For the intersection of the interior of the domain and the guard, we have to intersect the intervals $(-\infty..20)$ and $[20..\infty)$. Obviously, the intervals do not overlap in any point. The intersection is empty.

Now, we have to examine the next transition in the cycle. Again, we consider the guard of the transition, here from the *off* state to the *on* state, as well as the interior of the domain of the *off* state. The guard is $t <= 15$. The interval of the continuous variable t is $(-\infty..15]$. For the domain, we have the restrictions $t >= 15$ and $t >= 20$. The interior of the domain has the interval $(15..\infty)$ for the continuous variable t. Therefore, the intersection is empty as well.

The function *printIntersection* prints the result of the intersection of the interior of the domain and the guard for each cycle. Every cycle is evaluated in a separate list of the result. In the current case, there is only one cycle.

**Zeno> printIntersection "air.txt"*
[[]]

Looking at the outcome of the *printIntersection* function, we see that the tool calculates the intersection correctly.

Next, we evaluate the second part of the assumption, $\cap_{i=1}^{K-1} \partial D(q_i) = \emptyset$. The boundary is defined as the closure without the interior. Based on the fact that the closure of t is $(-\infty..20]$, while the interior is $(-\infty..20)$, the boundary of the domain of the *on* state is [20]. 20 is the only point that is in the closure but not in the interior. Thus, the boundary consists only of the element 20. Following the same argumentation, with the closure of the domain of the *off* state being $[15..\infty)$, the boundary of the domain of the *off* state is [15]. The intersection of the boundaries of the domain is empty since 15 and 20 do not intersect anywhere. To check the boundary intersection computed by the tool, the printIntersectBoundaries function is used.

**Zeno> printIntersectBoundaries "air.txt"*
[[]]

The boundary intersection works as expected for the air conditioning system. As the assumptions of Corollary 1 hold, we conclude that the automaton is zeno free. The main function runs the whole implementation and correctly computes that no zeno behavior can occur.

**Zeno> main "air.txt"*
The cycle [off on off] does not show zeno behavior.

In this section we have shown that the tool works for the air conditioning system as expected. In the next section, we will examine the example of the bouncing ball.

6.2 Bouncing Ball

The second example is the bouncing ball system as an example of an hybrid automaton containing a cycle with a *non-expanding* reset map, which shows zeno behavior. The input file of Fig. 6.2 shows how the bouncing ball system from Fig. 3.1 is translated into our grammar.

```
1 Automaton;
2 State, air, True, x >= 0, x = v;
3 Transition, air, air, x == 0, 0.9;
```

Figure 6.2: Input File for the Bouncing Ball System

The automaton consists of one cycle, which the tool detects.

**Zeno> printCycle "ball.txt"*
[["air","air"]]

Due to the fact that the cycle possesses a *non-expanding* reset map, we have to check it using Theorem 1. To use Theorem 1, we have to check that $G(q, q') \cap D(q)^0 = \emptyset, \forall (q, q') \in E$ holds. For this example with one transition, we only have to check the intersection of one guard with the interior of the *air* state. Having $x == 0$ as guard, the continuous variable x is not restricted to an interval but a single point, [0]. The domain has the restrictions $x >= 0$ and $x == 0$. Therefore, the interior of the domain has the interval $(0..\infty)$ for the continuous variable x. The intersection of the guard and the interior of the domain is empty, which *printIntersection* confirms:

**Zeno> printIntersection "ball.txt"*
[[]]

Based on Theorem 1, the automaton shows zeno behavior and the zeno set consists of the boundaries of the domains for each state in the cycle. The boundary of the domain from the *air* state is 0, since the interior is $(0..\infty)$, while the closure is $[0..\infty)$.

**Zeno> main "ball.txt"*
The zeno set for the cycle [air air] is:
In state "air" for the continuous variable x the zeno point(s) are: 0.0.

Note that the zeno point, 0.0 is printed as a float. The tool calculates the same result as we did by hand. We can record that the tool also works correctly for the example of the bouncing ball. Hence, the tool works so far as expected.

6.3 Water Tank

One example is left, the water tank system. Fig. 2.3 shows a model of the water tank system. The hybrid automaton of the system possesses an *identity* reset map and shows zeno behavior. The input file, which represents the automaton from Fig. 2.3, for the tool is pictured in Fig. 6.3.

```
1 Automaton;
2 State, q1, True, x2 >= r2, x1 = w - v1, x2 = - v2;
3 State, q2, True, x1 >= r1, x1 = - v1, x2 = w - v2;
4 Transition, q1, q2, x2 <= r2, 1;
5 Transition, q2, q1, x1 <= r1, 1;
```

Figure 6.3: Input File for the Water Tank System

Let us also derive the evaluation first by hand and then see if the tool works as expected. The automaton consists of one cycle with the state q_1 and state q_2.

**Zeno> printCycle "watertank.txt"*
[["q1","q2","q1"]]

Based on Corollary 1, we have to check $G(q,') \cap D(q)^0 = \emptyset, \forall (q, q') \in E$. Due to the guard $x2 <= r2$ of the transition from the state q_1 to state q_2, we get an interval of $(-\infty..r_2]$ for the continuous variable x_2. The interior of the domain of state q_1 is derived by the restrictions $x2 >= r2$ and $x1 <= r1$. Therefore, the continuous variable x_1 has the interval $(-\infty..r_1)$ and the interval of x_2 is $(r_2..\infty)$ for the interior of the domain. Taking the intersection of the guard with the interior of the domain

from the state q_1, we see that there is no overlap for x_2. Hence, the intersection is empty.

Further, we have to check the intersection of the interior of the domain from the state q_2 with the guard of the transition from state q_2 to q_1. The guard $x1 <= r1$ implies that the continuos variable x_1 has the interval $(-\infty..r_1]$. Based on the restrictions $x1 >= r1$ and $x2 <= r2$, the interior consists of the following intervals: $(r_1..\infty)$ for x_1 and $(-\infty..r_2)$ for x_2. Taking the intersection of guard and interior of the domain, we see that there is no overlap for x_1. Again, the intersection is empty. The fact that the intersection of the guard with the interior of the domain is empty for every transition pair yields that the overall intersection is empty as well.

**Zeno> printIntersection "watertank.txt"*
[[]]

Finally, we have to check $\cap_{i=1}^{K-1} \partial D(q_i) = \emptyset$. For the boundary, we need the closure as well as the interior. We already know the interiors. Therefore, we only need to compute the closures. The closure of the domain of state q_1 is $(-\infty..r_1]$ for x_1 and $[r_2..\infty)$ for x_2. For the domain of the state q_2, the closure is $[r_1..\infty)$ for x_1 and $(-\infty..r_2]$ for x_2. Hence, the boundaries for q_1 are r_1 for x_1 and r_2 for x_2. The state q_2 has the same boundaries. More formally said, $\partial D(q_1) = \begin{pmatrix} r_1 \\ r_2 \end{pmatrix}$ and $\partial D(q_2) = \begin{pmatrix} r_1 \\ r_2 \end{pmatrix}$. The intersection of the boundaries of the two states is not empty, but consists of r_1 for x_1 and r_2 for x_2.

**Zeno> printIntersectBoundaries "watertank.txt"*
[[("q1",[("x1",["r1"]),("x2",["r2"])]),("q2",[("x1",["r1"]),("x2",["r2"])])]]

Unfortunately, Corollary 1 does not hold true. Thus, the automaton can show zeno behavior. Its zeno set consists of the intersection of the boundaries.

**Zeno> main "watertank.txt"*
The zeno set for the cycle [q1 q2 q1] is:
In state "q1" for the continuous variable x1 the zeno point(s) are: r1; for the continuous variable x2 the zeno point(s) are: r2;
In state "q2" for the continuous variable x1 the zeno point(s) are: r1; for the continuous variable x2 the zeno point(s) are: r2.

So far, we have shown that the tool works correctly for every example that was presented in the thesis. Therefore, the tool works so far as expected. However, by now, we only know that it works for three examples.

6.4 Another Air Conditioning System

So far we tested 3 cases of the Table 4.1. However, the case of an overlapping interval is still missing. We will present one example for an overlapping interval in this section. The air conditioning automaton described in the input file represented by Fig. 6.4 is a slightly modified version from Henzinger [9].

```
1 Automaton;
2 State, on, True, x >= 18, x=-0.1x;
3 State, off, False, x <= 22, x=5-0.1x;
4 Transition, on, off, x > 21, 1;
5 Transition, off, on, x < 21.1, 1;
```

Figure 6.4: Input File for another Air Conditioning System

Like the air conditioning system automaton from Section 6.1, the new one has one cycle containing the *on* and *off* states, too.

**Zeno> printCycle "anotherAir.txt"*
[["on","off","on"]]

Taking the intersection of the guard of the transition from the *on* to the *off* state with the interior of the domain from the *on* state, we get an overlapping interval as the result. The guard has $x > 21$ as restriction, the interval for x is $(21..\infty)$. The domain consists of the restrictions $x >= 18$ and $x < 21.1$. Therefore, the interval for x is (18.0..21.1). As one can see, the intersection overlaps in (21.0..21.1).

For the other pair of guard and domain, we have the following intervals: The continuous variable x has $(-\infty..21.1)$ as interval for the guard. In the domain, x has the interval (21.0..22.0). This combination of guard and interior of domain results in the overlap (21.0..21.1).

**Zeno> printIntersection "anotherAir.txt"*
[[("on",["(21.0,21.1)"]),("off",["(21.0,21.1)"])]]

Due to the fact that the intersection of guard and interior of domain is not empty for all transitions of the cycle, we can not give a clear statement about zenoness.

**Zeno> main "anotherAir.txt"*
The overlapping intervals for the cycle [on off on] are:
For state "on" the overlapping interval(s) are:
For the continuous variable x (21.0,21.1);

For state "off" the overlapping interval(s) are:
For the continuous variable x (21.0,21.1).
Thus, no clear statement about zenoness can be made.

Besides the examples where we could clearly say whether the automaton can show zeno behavior, we now also know that the tool correctly detects an overlapping interval. In the next section, we will conduct a completeness evaluation.

6.5 Completeness

Cycle	I	NE	$G(q,q\prime) \cap D(q)^0 = \emptyset$	Zeno Set	Zeno Behavior	Test	Tested
×	−	×	×	$x_i \in \partial D(q_i)$	YES	ball	✓
×	do not care		−	No clear statement	MAYBE	anotherAir	✓
×	×	−	×	$\cap_{i=1}^{K-1} \partial D(q_i) = \emptyset$	NO	air	✓
×	×	−	×	$\cap_{i=1}^{K-1} \partial D(q_i)$	YES	watertank	✓
-	do not care			$\emptyset$	NO	noCycle	✓

Table 6.1: Tests for all Conditions for Zeno Behavior

Table 6.1 shows that every case of zeno behavior is covered by a test. In the previous sections, we have had a look at each test but the simple case of no cycle. To conduct our completeness evaluation, we need to test that every computation step is performed correctly. We start with the retrieval of the domain.

Table 6.2 shows all possible combinations of restrictions for a continuous variable in a state and the guard of its incoming transition in the cycle. Further, the table shows a test for each of these possibilities. Overall, 36 tests were conducted for retrieving the interior of the domain. All of the tests turned out fine.

For testing all combinations, it was crucial to make a distinction between a bound by a variable or a constant. The distinction is needed, since bounds by variables and by constants are treated in different branches. Due to the assumption that the boundaries of every continuous variable of a guard have to lie inside the boundaries of the destination state, it is vital to make sure that the domain is retrieved correctly. Table 6.2 shows that the tool retrieves the domain correctly using the assumption.

Knowing that the domains are retrieved correctly by the tool, the intersection of the interior of the domain with the guard is the next component of interest. First, we will consider the case of having variables as bounds in the restrictions. The intersection of the guard with the interior of the domain is taken for each continuous variable separately. For the intersection, it is necessary that the tool goes through all restrictions on the domain and finds the matching boundary variables. While going

State Restriction			Incoming Guard			Tests				
Lower	Upper	Equal	Lower	Upper	Equal	Variable		Constant		Interval
×	×	–	do not care			stateLower-AndUpper	✓	OverlapConst-Multiple-RestrictionEach	✓	$(l_1..u_1)$
–	–	–	×	×	–	OverlapVar-Multiple-RestrictionEach	✓	multOnOne-ContConst	✓	$(l_2..u_2)$
–	–	–	–	×	–	OverlapVar-Multiple-RestrictionEach, watertank	✓	multOnOne-ContConst	✓	$(-\infty..u_2)$
–	–	–	×	–	–	OverlapVar-Multiple-RestrictionEach	✓	multOnOne-ContConst	✓	$(l_2..\infty)$
–	–	–	–	–	×	equalVar	✓	equalConst	✓	[]
–	–	×	–	–	×	equalVar	✓	equalConst	✓	[]
–	–	×	–	–	–	equalVar	✓	equalConst	✓	[]
–	–	–	–	–	–	equalVar	✓	equalConst	✓	$(-\infty..\infty)$
–	×	–	×	×	–	multOnOne-ContVar	✓	multOnOne-ContConst	✓	$(l_2..u_1)$
–	×	–	×	–	–	OverlapVar-Multiple-RestrictionEach	✓	anotherAirCond, OverlapConst-One-RestrictionEach	✓	$(l_2..u_1)$
–	×	–	–	×	–	airGeneral, test3	✓	air	✓	$(-\infty..u_1)$
–	×	–	–	–	–	NoOverlap-VarMultiple-RestrictionEach	✓	NoOverlap-ConstMultiple-RestrictionEach	✓	$(-\infty..u_1)$
–	×	–	–	–	×	equalVar	✓	equalConst	✓	$(-\infty..u_1)$
×	–	–	×	×	–	multOnOne-ContVar	✓	multOnOne-ContConst	✓	$(l_1..u_2)$
×	–	–	×	–	–	airGeneral, NoOverlap-VarMultiple-RestrictionEach, test3	✓	air, NoOverlap-ConstMultiple-RestrictionEach	✓	$(l_1..\infty)$
×	–	–	–	×	–	OverlapVar-Multiple-RestrictionEach	✓	anotherAirCond, OverlapConst-One-RestrictionEach	✓	$(l_1..u_2)$
×	–	–	–	–	–	NoOverlap-VarMultiple-RestrictionEach	✓	NoOverlap-ConstMultiple-RestrictionEach	✓	$(l_1..\infty)$
×	–	–	–	–	×	equalVar	✓	equalConst, ball	✓	$(l_1..\infty)$

Table 6.2: Retrieving the Interval of the Interior of a Domain for a Continuous Variable with Tests

through all restrictions, we make use of the assumption that the boundaries of the guard of the incoming transition lie inside the boundaries of the state restrictions. Therefore, even if a matching variable retrieved from the incoming guard is not used for the generation of the interval of the domain, that variable can be used for the intersection.

In case a continuous variables is bounded by different variables in the guard and the domain, we can not always give a clear statement about the intersection. The variable bounding the continuous variable in the guard has to overlap in at least one point with the domain. Let us consider an example and focus on the continuous variable x_1. Further assume that x_1 has the interval, $[-\infty..r_1]$ and the guard the interval $[r_2..\infty]$. Here, $r_1 = r_2$ has to hold true. Otherwise, the guard never could hold true and the transition could never be triggered. If the overlap is bigger than exactly one point the intersection would never be empty. An example is $r_1 = r_2 + 3$. However, the boundaries of the continuous variable in domain and guard could be equal. If the boundaries are equal, $r_1 = r_2$, the intersection is empty for some cases. Hence, we can not give a clear statement in case we can not match the bounding variables.

Domain			Guard			Tests		
Lower	Upper	Equal	Lower	Upper	Equal	Variable		Intersection
r_1	r_2	–	r_1	r_2	–	test4	✓	$(r_1..r_2)$
r_1	r_2	–	–	r_2	–	test5	✓	$(r_1..r_2)$
r_1	r_2	–	r_1	–	–	test4	✓	$(r_1..r_2)$
r_1	r_2	–	–	–	$r_1 \vee r_2$	test6	✓	[]
r_1	r_2	–	–	–	–	stateLower-AndUpper	✓	$(r_1..r_2)$
–	–	–	r_1	r_2	–	notInDomain	✓	$[r_1..r_2]$
–	–	–	–	r_2	–	notInDomain	✓	$(-\infty..r_2]$
–	–	–	r_1	–	–	notInDomain	✓	$[r_1..\infty)$
–	–	–	–	–	r_1	equalVar	✓	$[r_1]$
–	–	r_1	–	–	r_1	equalVar	✓	[]
–	–	r_1	–	–	–	equalVar	✓	[]
–	r_2	–	r_1	r_2	–	test7	✓	$[r_1..r_2)$
–	r_2	–	r_2	–	–	test7	✓	[]
–	r_2	–	–	r_2	–	airGeneral	✓	$(-\infty..r_2)$
–	r_2	–	–	–	–	notInDomain	✓	$(-\infty..u_1)$
–	r_2	–	–	–	r_2	equalVar	✓	[]
r_1	–	–	r_1	r_2	–	test7	✓	$(r_1..r_2]$
r_1	–	–	r_1	–	–	airGeneral	✓	$(r_1..\infty)$
r_1	–	–	–	r_1	–	ballVar	✓	[]
r_1	–	–	–	–	–	notInDomain	✓	$(r_1..\infty)$
r_1	–	–	–	–	r_1	equalVar	✓	[]

Table 6.3: Intersection of the Interior of a domain with a Guard for a Continuous Variable with Tests

Table 6.3 shows all possible combinations to intersect the interior of the domain with the guard for a continuous variable if a boundary variable matches. The tool draws the right conclusions for all of these cases. However, the tool does not always print the correct interval, but only the interval from the intersection, where the variables match. Therefore, the first case, having a lower and upper bound for both domain and guard, would not return the interval $(r_1..r_2)$, but $(-\infty..r_2)$ and $(r_1..\infty)$.

For state q1 the overlapping intervals are:
For the continuous variable x1 ("r1"..Infinity)(-Infinity.."r2");

For the purpose of our tool, it is sufficient to check if the intersection is empty. Based on the way the overlap is calculated, the intervals are not always exact. Nonetheless, the tool derives the right conclusion that the intersection is not empty. Further, the intervals give an indication how large the overlap actually is.

Guard			Tests		
Lower	Upper	Equal	Constant		Interval
r_1	r_2	–	multOnOne-ContConst	✓	$[r_1..r_2]$
–	r_2	–	air	✓	$(-\infty..r_2]$
r_1	–	–	air	✓	$[r_1..\infty)$
–	–	e_1	ball	✓	$[e_1]$
–	–	–	test8	✓	$(-\infty..\infty)$

Table 6.4: Interval for the Guard using Constants as Boundaries with Tests

Second, we will consider the case of having constants as bounds in restrictions. So far, we tested that the tool creates the interiors of the domain correctly. For the intersection with the guard, the correct creation of the guard is crucial. As one can see in Table 6.4, the tool also generates the guards correctly in case the continuous variables are bounded by a constant. Now, we know that we construct the intervals for the interior of the domain and the guard correctly. The intersection is left to be checked. Assuming the *Data.IntervalMap.Interval* library used implemented the overlap function correctly denotes that our intersection is correct. Further, for boundaries with constants, the tool generates the correct overlapping interval.

Table 5.3 states that the boundaries of the domain always consist of the bound that was used to create the interval. With relational operators, the bound is always in the closure of the interval, but never in the interior. Therefore, by the definition of the boundaries, they only consist of the bounds used to span the interval. As we retrieve the restrictions for the domain correctly, we can generate the boundaries correctly by simply taking the bounds of the restrictions used to generate the domain.

The only case left to check is the intersection of the boundaries of the domains. Here are two cases to consider: The first case is that every continuous variable of the given two domains intersect in at least one point. An example is the water tank system. As shown previously, the tool works correctly for the water tank example. The other case is that at least one continuous variable does not intersect. Thus, the intersection is empty. The air conditioning system from Section 6.1 shows that the tool handles the case correctly. Further, since our boundaries of the domains are correctly retrieved and we use the intersect function from Haskell, no problem should occur.

So far, the tool works correctly. However, we have a few more properties that the tools has to handle properly.

Feature	Test	Implemented
Incoming transitions, which are not in the cycle are not considered for the domain	OverlapConst-OneRestriction-EachAdditionalTrans, OverlapVarOneRestric-tionEachAdditionalTrans	✓
Order of defining lower and upper bounds does not matter	OverlapVar-MultipleRestrictionEach	✓
Order of defining transitions and states, except for the case of reflex-ive edges, does not matter	OverlapVarMultiple-RestrictionEach , threeCy-clesCombined	✓
Order of the continuous variables in the restriction does not matter	OverlapVarMultiple-RestrictionEach	✓
Combination of distinct cycles are correctly evaluated	watertankAndAir	✓
Compute the boundaries of the do-main correctly	test3	✓
Combinations of variables and con-stants in one restriction		↯
Improve the printout of the inter-secting interval for boundaries with variables	test4	↯
Always handle reflexive edges cor-rectly	threeCyclesCombined	↯

Table 6.5: Properties of the Tool

Table 6.5 shows the general properties of the tool. The first to fourth property is needed to ensure that the domain and the guard are always retrieved correctly. The algorithm for the cycle detection allows more than one cycle, which is checked by the fifth property. Property six ensures that while we intersect the interior of the domain with the guard, we go through all restrictions of the domain to match the bounding variable. Moreover, it is checked that the tool evaluates each cycle.

Unfortunately, there are also some features that are not completely supported. Based on the different ways of handling variables and constants as bounds in restrictions, we do not allow combinations of both in restrictions.

Reflexive edges are not handled correctly due to the way the cycles are detected. The only case a reflexive edge attached to another cycle is detected correctly, is the following: The state with the reflexive edge has to be evaluated last for detecting the cycles. This means that the state of the reflexive edge has to occur after every state of the cycle has occurred before as a source in a transition. The occurrence of transitions depends on the order in which the transitions are defined. The tool creates the mapping from state name to number based on the occurrence of that state as source in a transition. Since the reflexive edge is only evaluated correctly when it is the last entry in the adjacency matrix by the algorithm, the mapping of the state with the reflexive edge must have the highest mapping number. Therefore, it has to be the last to get a number assigned. To solve the problem with detecting reflexive edges correctly, probably another algorithm for detecting cycles is needed.

To sum this section up, the completeness evaluation shows that the tool works as desired. Only three cases are not yet completely implemented as wanted. Even without these cases, the tool can be used to validate Modelica simulations.

7 Future Work

Even though the tool to detect zeno sets works as desired, there is still some work to do. For once, the incomplete features from Table 6.5 should be improved and incorporated. Besides getting every feature to work as desired, a better integration with OpenModelica is desirable. After the demonstrated *offline* tool, an indication directly at runtime of the simulation would be the next step. However, before a complete integration, it might be easier to use the Modelica model as an input file and let the tool parse the Modelica model. The HyAuLib would be of great help for parsing the file. Another approach could be to leave the input file as it is, but generate the corresponding Modelica model automatically, using the HyAuLib.

However, besides a closer integration to OpenModelica, it is also important to handle the model correctly in case it shows zeno behavior. Johansson et al. [13] show an approach how to deal with an automaton that shows zeno behavior. For example, the water tank system had the flaw that no delay for the switching of the water source from one tank to the other was assumed. Implementing such a delay the water tank should work as desired. Fig. 7.1 represents the hybrid automaton of the water tank system including a delay. Using the approach described by Johansson et al. in [13] the modeler has to revisit the system and check if the model is missing a detail of the physical system. For the water tank example, the missing detail would be the delay.

Another approach is described in [10], where the authors describe a controller that regulates the hybrid automaton based on invariants. Using these invariants, one can define that every transition takes a certain amount of time. Thereby, we can ensure that zeno behavior can not occur. However, one has to be careful. For example, if a state expects the results instantly, a different problem than zeno behavior can occur, since the result would also be delayed. Nonetheless, a closer integration to OpenModelica should be of higher priority.

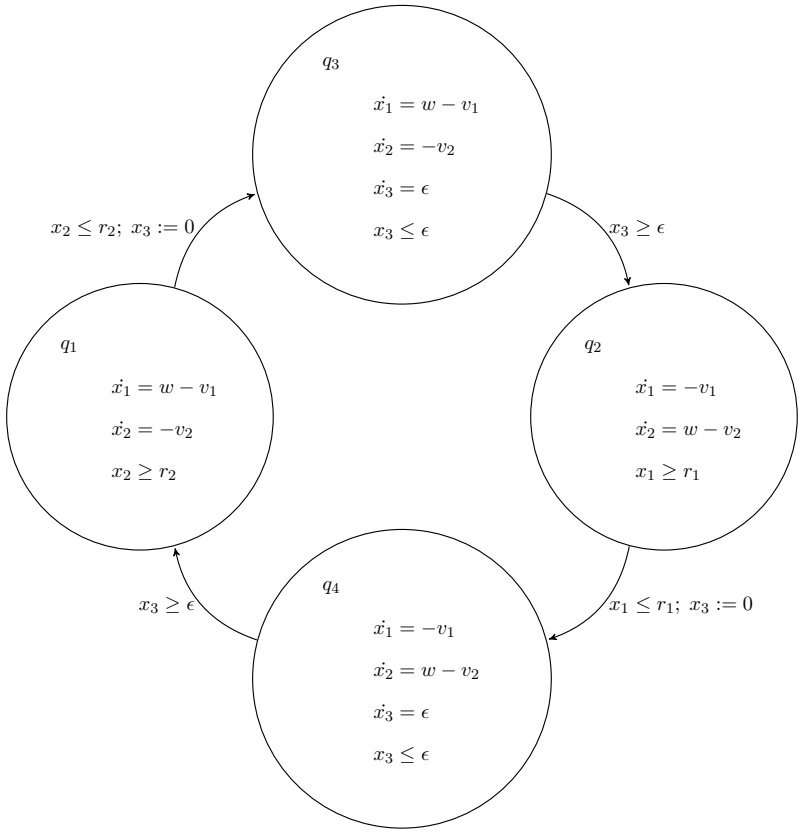

Figure 7.1: Hybrid Automaton of the Water Tank with Delay

8 Conclusion

In this thesis we derived a tool to detect zeno sets of hybrid systems. The zeno sets are of interest when validating Modelica simulations of hybrid systems. Validation is crucial since the outcome of the simulation can be falsified and therefore, differ from actually running the physical system. Zeno behavior can influence the result of simulating a hybrid system. Thus, knowing when zeno behavior can occur is of interest for validation of hybrid systems. However, the tool does not only determine if a hybrid automaton can show zeno behavior, but also the zeno points to which the automaton converges in case of a zeno execution.

Knowing the zeno points, the modeler does not have to check every simulation of a hybrid system, but only the ones converging to at least one zeno point. If the simulation does converge to a zeno point, the modeler still has to derive the correct result by hand or use one of the described approaches to handle zeno behavior. Nonetheless, having the tool is another safety net while validating a system by simulation.

A failure of such a hybrid system could have far-ranging impact on the environment. Hence, it is necessary to know that the simulation is trustworthy, while validating the system. Without knowing when the simulation outcome can be falsified, it is not possible to achieve a proper validation of the system.

Even though some features are not fully supported, the tool works the way it is desired and specified by Theorem 1 and Corollary 1 for the supported features. In conclusion, we can state that the tool can be used to detect zeno sets and improve our tool chain to validate hybrid systems.

Bibliography

[1] Alur, R., and Henzinger, T. A. Modularity for timed and hybrid systems. In *Proceedings of the 8th International Conference on Concurrency Theory* (London, UK, 1997), CONCUR '97, Springer-Verlag, pp. 74–88.

[2] Bauer, K. *A New Modelling Language for Cyber-Physical Systems*. PhD thesis, Department of Computer Science, University of Kaiserslautern, Germany, Kaiserslautern, Germany, January 2012.

[3] Benveniste, A., Caillaud, B., and Pouzet, M. The fundamentals of hybrid systems modelers. In *2010 49th IEEE Conference on Decision and Control (CDC)* (dec. 2010), pp. 4180 –4185.

[4] Ferreira, J., and Estima de Oliveira, J. Modelling hybrid systems using statecharts and modelica. In *ETFA '99. 1999 7th IEEE International Conference on Emerging Technologies and Factory Automation, 1999. Proceedings.* (1999), vol. 2, pp. 1063 –1069 vol.2.

[5] Fritzson, P. Modelica x2014; a cyber-physical modeling language and the OpenModelica environment. In *2011 7th International Wireless Communications and Mobile Computing Conference (IWCMC)* (july 2011), pp. 1648 –1653.

[6] Fritzson, P., Aronsson, P., Lundvall, H., Nyström, K., Pop, A., Saldamli, L., and Broman, D. The OpenModelica Modeling, Simulation, and Software Development Environment. *Simulation News Europe 44*, 45 (dec 2005).

[7] Fritzson, P., and Bunus, P. Modelica - a general object-oriented language for continuous and discrete-event system modeling and simulation. In *35th Annual Simulation Symposium, 2002. Proceedings.* (april 2002), pp. 365 – 380.

[8] Harel, D., and Pnueli, A. Logics and models of concurrent systems. Springer-Verlag New York, Inc., New York, NY, USA, 1985. On the development of reactive systems, pp. 477–498.

[9] Henzinger, T. The theory of hybrid automata. In *Eleventh Annual IEEE Symposium on Logic in Computer Science, 1996. LICS '96. Proceedings.* (jul 1996), pp. 278 –292.

[10] Heymann, M., Lin, F., Meyer, G., and Resmerita, S. Analysis of Zeno behaviors in a class of hybrid systems. *IEEE Transactions on Automatic Control 50*, 3 (march 2005), 376 – 383.

[11] Johansson, K., Lygeros, J., Sastry, S., and Egerstedt, M. Simulation of zeno hybrid automata. In *Proceedings of the 38th IEEE Conference on Decision and Control, 1999.* (1999), vol. 4, pp. 3538 –3543 vol.4.

[12] Johansson, K., Lygeros, J., Zhang, J., and Sastry, S. Hybrid automata: a formal paradigm for heterogeneous modeling. In *IEEE International Symposium on Computer-Aided Control System Design, 2000. CACSD 2000.* (2000), pp. 123 –128.

[13] Johansson, K. H., Egerstedt, M., Lygeros, J., and Sastry, S. On the regularization of zeno hybrid automata. *Systems & Control Letters 38*, 3 (1999), 141 – 150.

[14] Lygeros, J., Johansson, K., Simic, S., Zhang, J., and Sastry, S. Dynamical properties of hybrid automata. *IEEE Transactions on Automatic Control 48*, 1 (jan 2003), 2 – 17.

[15] Pulecchi, T., and Casella, F. Hyaulib: Modelling hybrid automata in Modelica hyaulib: modelling hybrid automata in modelica. *In Proceedings 6th International Modelica Conference, Bielefeld, Germany, Mar. 3-4, 2008* (2008), 239–246.

[16] Raymond, P., Nicollin, X., Halbwachs, N., and Weber, D. Automatic testing of reactive systems. In *The 19th IEEE Real-Time Systems Symposium, 1998. Proceedings.* (dec 1998), pp. 200 –209.

[17] Rinast, J. Enhancing Uppaal's explanatory power using static zeno run analysis. Diplomarbeit, TU Hamburg-Harburg, Apr. 2012.

[18] Schmitz, G., Milano, P. D., Elmqvist, D. H., Ab, D., Otter, M., erik Arzen, K., and Dressler, I. Stategraph – a modelica library for hierarchical state machines abstract. *In Proceedings of the 4th International Modelica Conference, Hamburg, Germany* (2005), 569–578.

[19] Tarjan, R. E. Enumeration of the elementary circuits of a directed graph. Tech. rep., Cornell University, Ithaca, NY, USA, 1972.

[20] Zhang, J., Johansson, K. H., Lygeros, J., and Sastry, S. Zeno hybrid systems. *International Journal of Robust and Nonlinear Control 11*, 5 (2001), 435–451.

Appendix

1 Test Files

air:

```
Automaton;
State, on, True, t <= 20, t = 1;
State, off, True, t >= 15, t = -1;
Transition, off, on, t <= 15, 1;
Transition, on, off, t >= 20, 1;
```

airGeneral:

```
Automaton;
State, on, True, t <= r1, t = 1;
State, off, True, t >= r2, t = -1;
Transition, on, off, t >= r1, 1;
Transition, off, on, t <= r2, 1;
```

anotherAir:

```
Automaton;
State, on, True, x >= 18, x=-0.1x;
State, off, False, x <= 22, x=5-0.1x;
Transition, on, off, x > 21, 1;
Transition, off, on, x < 21.1, 1;
```

anotherAirCond:

```
Automaton;
State, on, True, x >= 18, x=-0.1x;
State, off, False, x <= 22, x=5-0.1x;
Transition, on, off, x > 21, 1;
Transition, off, on, x < 19, 1;
```

ball:

```
Automaton;
State, air, True, x >= 0, x = v;
Transition, air, air, x == 0, 0.9;
```

ballVar:

```
Automaton;
State, air, True, x >= r, t = 1;
Transition, air, air, x <= r, 0.9;
```

equalConst:

```
Automaton;
State, q1, True, x1 <= 1 AND x2 >= 2 AND x3 == 3 AND x4 == 4, x1 = w
    - v1, x2 = - v2;
Transition, q1, q1, x1 == 1 AND x2 == 2 AND x3 == 3 AND x5 == 5, 0.9;
```

equalVar:

```
Automaton;
State, q1, True, x1 <= r1 AND x2 >= r2 AND x3 == r3 AND x4 == r4, x1
    = w - v1, x2 = - v2;
Transition, q1, q1, x1 == r1 AND x2 == r2 AND x3 == r3 AND x5 == r5,
    0.9;
```

multOnOneContConst:

```
Automaton;
State, q1, Tue, x1 <= 1 AND x2 >= 2, x1 = w - v1, x2 = - v2;
Tansition, q1, q1, x1 <= 5 AND x1 >= 6 AND x2 <= 7 AND x2 >= 8 AND x3
    <= 3 AND x3 >= 4 AND x4 <= 9 AND x5 <= 10, 0.9;
```

multOnOneContVar:

```
Automaton;
State, q1, True, x1 <= r1 AND x2 >= r2, x1 = w - v1, x2 = - v2;
Transition, q1, q1, x1 <= r5 AND x1 >= r6 AND x2 <= r7 AND x2 >= r8
    AND x3 <= r3 AND x3 >= r4 AND x4 <= r9 AND x5 <= r10, 0.9;
```

noCycle:

```
Automaton;
State, on, True, t <= 20.0, t = 1;
State, off, True, t >= 15, t = -1;
Transition, on, off, t >= 20, 1;
```

NoOverlapConstMultipleRestrictionEach:

```
Automaton;
State, q1, True,    x2 >= 20 AND x3 >= 30, x1 = w - v1, x2 = - v2;
State, q2, True,    x1 <= 10 AND x4 >= 40, x1 = - v1, x2 = w - v2;
Transition, q1, q2, x2 <= 20 AND x4 >= 40, 0.9;
Transition, q2, q1, x1 >= 10 AND x3 >= 30, 1;
```

NoOverlapConstOneRestrictionEach:

```
Automaton;
State, q1, True,    x2 >= 20, x1 = w - v1, x2 = - v2;
State, q2, True,    x1 >= 10, x1 = - v1, x2 = w - v2;
Transition, q1, q2, x2 <= 20, 0.9;
Transition, q2, q1, x1 <= 10, 1;
```

NoOverlapVarMultipleRestrictionEach:

```
Automaton;
State, q1, True,    x2 >= r2 AND x3 >= r3, x1 = w - v1, x2 = - v2;
State, q2, True,    x1 >= r1 AND x4 <= r4, x1 = - v1, x2 = w - v2;
Transition, q1, q2, x2 <= r2 AND x4 <= r4, 1;
Transition, q2, q1, x1 <= r1 AND x3 >= r3, 1;
```

NoOverlapVarOneRestrictionEach:

```
Automaton;
State, q1, True,    x2 >= r2, x1 = w - v1, x2 = - v2;
State, q2, True,    x1 >= r1, x1 = - v1, x2 = w - v2;
Transition, q1, q2, x2 <= r2, 0.9;
Transition, q2, q1, x1 <= r1, 1;
```

notInDomain:

```
Automaton;
State, q1, True, x2 <= r2, x1 = w - v1, x2 = - v2;
State, q2, True, x4 >= r4, x1 = - v1, x2 = w - v2;
Transition, q1, q2, x3 <= r3 AND x3 >= r5 AND x5 <= r6 AND x6 >= r7,
    1;
Transition, q2, q1, x1 <= r1, 1;
```

OverlapConstMultipleRestrictionEach:

```
Automaton;
State, on, True, x >= 18 AND x < 24, x=-0.1x;
State, off, False, x <= 23 AND x > 0, x=5-0.1x;
Transition, on, off, x > 21 AND x < 23, 1;
Transition, off, on, x < 21 AND x > 18, 1;
```

OverlapConstOneRestrictionEach:

```
Automaton;
State, on, True, x >= 18, x=-0.1x;
State, off, False, x <= 22, x=5-0.1x;
Transition, on, off, x > 21, 1;
Transition, off, on, x < 21.1, 1;
```

OverlapConstOneRestrictionEachAdditionalTrans:

```
Automaton;
State, on, True, x >= 18 AND x1 >= 20, x=-0.1x;
State, off, False, x <= 22, x=5-0.1x;
State, q1, False, x <= 22, x=5-0.1x;
Transition, on, off, x > 21, 1;
Transition, off, on, x < 21.1, 1;
Transition, q1, on, x1 >= 20, 1;
```

OverlapVarMultipleRestrictionEach:

```
Automaton;
State, q1, True,    x2 >= r2 AND x1 >= r1 AND x3 >= r3, x1 = w - v1,
    x2 = - v2;
State, q2, True,    x1 <= r1 AND x3 <= r3 AND x2 <= r2, x1 = - v1, x2
    = w - v2;
Transition, q1, q2, x3 >= r3 AND x2 >= r2 AND x1 >= r1 AND x4 >= r4,
    0.9;
```

```
5 Transition, q2, q1, x2 <= r2 AND x3 <= r3 AND x1 <= r1 AND x5 <= 1
    AND x6 >= 0 AND x6 <= 1, 1;
```

OverlapVarOneRestrictionEachAdditionalTrans:

```
1 Automaton;
2 State, q1, True,    x2 >= r2 AND x3 >= r3, x1 = w - v1, x2 = - v2;
3 State, q2, True,    x1 >= r1, x1 = - v1, x2 = w - v2;
4 Transition, q1, q2, x2 >= r2, 0.9;
5 Transition, q2, q1, x1 >= r1, 1;
6 Transition, q3, q1, x3 <= r3, 1;
```

stateLowerAndUpper:

```
1 Automaton;
2 State, q1, True, x1 <= r1 AND x1 >= r2, x1 = w - v1, x2 = - v2;
3 Transition, q1, q1, x2 <= r1, 0.9;
```

test3:

```
1 Automaton;
2 State, q1, True, x1 <= r1, x1 = w - v1, x2 = - v2;
3 State, q2, True, x1 >= r1, x1 = - v1, x2 = w - v2;
4 Transition, q1, q2, x1 >= r2, 0.9;
5 Transition, q2, q1, x1 <= r2, 1;
```

test4:

```
1 Automaton;
2 State, q1, True, x1 >= r1 AND x1 <= r2, x1 = w - v1, x2 = - v2;
3 State, q2, True, x1 >= r1, x1 = - v1, x2 = w - v2;
4 Transition, q1, q2, x1 >= r1 AND x1 <= r2, 0.9;
5 Transition, q2, q1, x1 >= r1, 1;
```

test5:

```
1 Automaton;
2 State, q1, True, x1 >= r1 AND x1 <= r2, x1 = w - v1, x2 = - v2;
3 Transition, q1, q1, x1 <= r2, 0.9;
```

test6:

```
Automaton;
State, q1, True,  x2 >= r3 AND x2 <= r4, x1 = w - v1, x2 = - v2;
Transition, q1, q1,  x2 == r3, 0.9;
```

test7:

```
Automaton;
State, q1, True, x1 <= r1 AND x2 <= r3, x1 = w - v1, x2 = - v2;
State, q2, True, x3 >= r4, x1 = - v1, x2 = w - v2;
Transition, q1, q2, x1 <= r1 AND x1 >= r2 AND x2 >= r3, 1;
Transition, q2, q1, x3 >= r4 AND x3 <= r5, 1;
```

test8:

```
Automaton;
State, q1, True, x1 <= r1 AND x2 <= r3, x1 = w - v1, x2 = - v2;
Transition, q1, q1, x1 <= r1 AND x1 >= r2, 1;
```

threeCyclesCombined:

```
Automaton;
State, q1, True, x1 >= r1, x1 = w - v1, x2 = - v2;
State, q2, True, x1 >= r1, x1 = - v1, x2 = w - v2;
State, q3, True, x1 >= r1, x1 = w - v1, x2 = - v2;
Transition, q1, q2, x1 <= r1, 0.9;
Transition, q3, q3, x1 <= r1, 1;
Transition, q2, q1, x1 <= r1, 1;
Transition, q3, q2, x1 <= r1, 1;
Transition, q2, q3, x1 <= r1, 0.9;
```

watertank:

```
Automaton;
State, q1, True, x2 >= r2, x1 = w - v1, x2 = - v2;
State, q2, True, x1 >= r1, x1 = - v1, x2 = w - v2;
Transition, q1, q2, x2 <= r2, 1;
Transition, q2, q1, x1 <= r1, 1;
```

watertankAndAir:

```
Automaton;
```

```
State, q1, True, x2 >= r2, x1 = w - v1, x2 = - v2;
State, q2, True, x1 >= r1, x1 = - v1, x2 = w - v2;
Transition, q1, q2, x2 <= r2, 1;
Transition, q2, q1, x1 <= r1, 1;
State, on, True, t <= 20, t = 1;
State, off, True, t >= 15, t = -1;
Transition, on, off, t >= 20, 1;
Transition, off, on, t <= 15, 1;
```

2 Haskell Code

2.1 Defined Data Structures

```
> data Automata = Automaton [States] [Transitions] deriving (Show)

> data States = State Name Start [Condition] [Equation] deriving (Show)

> data Transitions = Trans Source Dest [Condition] Reset deriving(Show)

> type Name = [Char]

> data Condition = Cond Variable Comp Variable deriving (Show, Eq)
> data Variable = Var1 Float | Var2 Name deriving (Show, Eq)
> type Comp = [Char]

> type Reset = Float

> type Start = Bool

> type Equation = [Char]

> type Dest = Name
> type Source = Name

> type AdjacentEntry = (Source, [Dest])
> type AdjacentList = [(Source,Int,[Dest])]

> type Mapping = [(Name, Int)]
> type AdjaList = [(Int,[Int])]
> type AdjaRow = (Int,[Int])

> type Cycle = [Int]
> type Cycles = [Cycle]
> type NameCycle = [Name]
> type NameCycles = [[Name]]
> type Marked = [Int]
```

2.2 State and Transition Parsing

```
States = State Name Start Condition [Equation]
Here all elements of a State are parsed into the data structure.
Further, the expected input data is matched against the data structure.

> getState :: [[Char]] -> [States]
> getState (x:xs:xss:xsss:xssss) = [(State xs (getStart xss) (getConds xsss)
    xssss)]
> getState _ = error "Not correctly defined state"

Transitions = Trans Source Dest Condition Reset
Here all elements of a Transition are parsed into the data structure.
Further, the expected input data is matched against the data structure.

> getTrans :: [[Char]] -> [Transitions]
> getTrans (x:xs:xss:xsss:xssss:xssssss) = [(Trans xs xss (getConds xsss)
    (getReset xssss))]
> getTrans _ = error "Not correctly defined transition"
```

2.3 Cycle Detection

```
Here the algorithm of Robert Tarjan is applied for "Enumeration of the
    Elementary Circuits of a directed graph" is applied.
findCycle applies the backtracking algortihm for every adcacency list.

> findCycles :: AdjaList -> Cycles
> findCycles [] = []
> findCycles ((a,v):xs)
>                   | null (backTrack v a [a] xs) = findCycles xs
>                   | otherwise = backTrack v a [a] xs : findCycles xs

The backTrack function implements the algorithm of Robert Tarjan as specified
    in the paper.

> backTrack ::  [Int] -> Int -> [Int] -> AdjaList -> [Int]
> backTrack [] _ _ _ = []
> backTrack (v:vs) s marked x
>                            | v < s = backTrack vs s marked x
>                            | v == s = s : marked ++ backTrack vs s marked x
>                            | not (elem v marked) = backTrack (getAdjaRow x
    v) s (v : marked) x
>                            | otherwise = backTrack vs s marked x

> getAdjaRow :: AdjaList -> Int -> [Int]
> getAdjaRow [] _ = []
> getAdjaRow ((a,b):xs) s
>                   | a == s = b
>                   | otherwise = getAdjaRow xs s
```

2.4 Retrieving Restrictions on Domain

```
Returns the Conditions for a given State Name.

> retrieveDomain :: Automata -> Name -> [Condition]
> retrieveDomain (Automaton x _) y = getStateDomain (findState x y)
>   where
>       findState [] v = error ("State " ++ show v ++ " is not defined")
>       findState (u:us) v
>                   | v == (getStateName u) =  u
>                   | otherwise = findState us v

Returns all Conditions from the Transition that have the Destination and
    Source given by the input.
Based on the properties only one Transition can have that pair of Destination
    and Source, the tool can stop after one match was found and return the
    restrictions of that Guard.

> retrieveTransToState :: Automata -> Dest -> Source -> [Condition]
> retrieveTransToState (Automaton _ x) y z = helper x y z
>   where
>       helper [] _ _  = []
>       helper (x:xs) y z
>           | getDestination x == y &&  getSource x == z = (getTransGuard x)
>           | otherwise = helper xs y z
```

2.5 Intersection

```
Checks if there is an empty list in a list

> anyNull :: [[a]] -> Bool
> anyNull x = any null x

We pairwise take an intersection of the interior of the Domain with the Guard.
We do this recursively until we visited all States in a Cycle.
As long as in one variable/space the domain and the guard do not intersect,
    they do not intersect at all.

> takeIntersection :: Automata -> NameCycle -> Name -> [(Source, [[Char]])]
> takeIntersection x y z = helper x y [] z
>   where
>       helper _ [b] c z = c
>       helper a (b:bs:bss) c z
>                       | anyNull (checkTransition b a ((retrieveDomain a b)
    ++ (retrieveTransToState a b z)) (retrieveGuard a b bs)) = helper a
    (bs:bss) c b
>                       | otherwise = (helper a (bs:bss) (c  ++ [(b ,
    (checkTransition b  a ((retrieveDomain a b) ++ (retrieveTransToState a b
    z)) (retrieveGuard a b bs))])) b

```

```
Here the intersection of guard with interior of domain is done for each continuous variable separately.
For the intersection of a continuous variable, we retrieve all restrictions, which use that continuous variable and remove these restrictions.
First we use the continuous variable of the first restriction in the guard and do this recursively until no restrictions are left.
Now, the tool checks if there are restrictions left in the domain and evaluates them.

> checkTransition :: Source  -> Automata -> [Condition] -> [Condition] -> [[Char]]
> checkTransition _ _ [] [] = []
> checkTransition w x (y:ys) [] = checkDomain y : checkTransition w x ys []
> checkTransition w x y ((Cond (Var2 a) b c):zs) =  case helper w x y ((Cond (Var2 a) b c): getAllRelevantCondsFromTrans a zs) of
>                   [] -> [] : checkTransition w x (y \\ (getAllRelevantCondsFromDomain a y))  (zs \\ (getAllRelevantCondsFromTrans a zs))
>                   u  -> ("For the continuous variable " ++ a ++ " " ++ u) : checkTransition w x (y \\ (getAllRelevantCondsFromDomain a y))  (zs \\ (getAllRelevantCondsFromTrans a zs))
>    where
>        helper w x y ((Cond (Var2 a) b c): zs) = (intersectIntervalls (getAllRelevantCondsFromDomain a y) ((Cond (Var2 a) b c): zs) a w)

In case a continuous variable is only restricted in the Domain, we format the overlap here.

> checkDomain :: Condition ->  [Char]
> checkDomain (Cond (Var2 a) b c)
>           | b == "==" = []
>           | (b == "<" || b == "<=") = "For the continuous variable " ++ a ++" (" ++ show ninfi ++ ".." ++ getVarName c ++")"
>           | (b == ">" || b == ">=") = "For the continuous variable " ++ a ++ "( " ++ getVarName c ++ ".." ++ show infi ++")"

Now we check how many restrictions we have on one continuous variable and based on it proceed with intersecting them.

> intersectIntervalls :: [Condition] -> [Condition] -> [Char] -> Source -> [Char]
> intersectIntervalls [] [] y z = []
> intersectIntervalls [] ((Cond (Var2 a) b c): zs) y z
>           | b == "==" = "[" ++ getVarName c ++"]" ++ intersectIntervalls [] zs y z
>           | (b == "<" || b == "<=") = "(" ++ show ninfi ++ ".." ++ getVarName c ++")" ++ intersectIntervalls []  zs y z
>           | (b == ">" || b == ">=") = "(" ++ getVarName c ++ ".." ++ show infi ++")" ++ intersectIntervalls []  zs y z
> intersectIntervalls x y z w = checkIntervalLenght x y z w

Based on the number of restrictions we call a different method to generate the intervals and then intersect.

> checkIntervalLenght :: [Condition] -> [Condition] -> Name -> Source -> [Char]
> checkIntervalLenght [x] [y] z _ = checkInterval x y z
```

```
54 > checkIntervalLenght (x:xs:xss) [y] z w =
       checkIntervalWithMultipleRestrictionsOnDomain (x:xs:xss) y z
55 > checkIntervalLenght [x] (y:ys:yss) z w
56 >                   | null yss = checkIntervalWithMultipleRestrictionsOnGuard
       x (y:ys:yss) z
57 >                   | otherwise = error ("Multiple restriction on" ++ show z
       ++ "guard in state " ++ show w)
58 > checkIntervalLenght (x:xs:xss) (y:ys:yss) z w
59 >                   | null yss = checkIntervalWithMultipleRestrictions
       (x:xs:xss) (y:ys:yss) z
60 >                   | otherwise = error ("Multiple restriction on" ++ show z
       ++ "guard in state " ++ show w)
61
62 The following functions create the interior of the domain and the set of the
       guard for the intersection.
63
64 Here we have the case of multiple restrictions only on the domain of one
       continuous variables using variables as bounds.
65 The domain can have more than two restrictions, due to the fact that incoming
       transitions are considered as well.
66 For the domain we use the assumption that the boundaries of the guard from
       the incoming transition always lays in the boundaries of the domain of
       the state.
67 Therefore, we look through every restriction on the domain and see if it can
       be matched against the variable of the guard.
68
69 > checkIntervalWithMultipleRestrictionsOnDomain :: [Condition] -> Condition
       -> Name -> [Char]
70 > checkIntervalWithMultipleRestrictionsOnDomain ((Cond _ a (Var2 b)) : (Cond
       _ c (Var2 d)) : xs) (Cond g e (Var2 f)) z
71 >       | b == f = compareInterval a e f
72 >       | d == f = compareInterval c e f
73 >       | null xs = error ("Could not compare " ++ show b ++ " and" ++ show d
       ++ " and " ++ show f ++ " in state " ++ show z)
74 >       | (length xs == 1) = checkInterval (head xs) (Cond g e (Var2 f)) z
75 >       | otherwise = checkIntervalWithMultipleRestrictionsOnDomain xs (Cond
       g e (Var2 f)) z
76
77 Here we have the case of multiple restrictions only on the domain of one
       continuous variables using constatns as bounds.
78 The domain can have more than two restrictions, due to the fact that incoming
       transitions are considered, as well.
79 For the domain we use the assumption that the boundaries of the guard from
       the incoming transition always lays in the boundaries of the domain of
       the state.
80 Therefore, we look through every restriction on the domain and see if it can
       be matched against the constants of the guard.
81
82 > checkIntervalWithMultipleRestrictionsOnDomain ((Cond g a (Var1 b)) : (Cond
       _ c (Var1 d)) : xs) (Cond h e (Var1 f)) z
83 >       | (a == "==" || c == "==") = []
84 >       | a == c && b == d = checkGivenIntervalls (createInteriorInterval a
       b) (createInterval e f)
85 >       | (a == "<=" || a == "<") && (c == ">=" || c == ">") =
       checkGivenIntervalls (createInteriorIntervalWithTwoBounds d b)
       (createInterval e f)
86 >       | (a == ">=" || a == ">") && (c == "<=" || c == "<") =
       checkGivenIntervalls (createInteriorIntervalWithTwoBounds b d)
```

```
    (createInterval e f)
87 >        | (a == ">=" || a == ">") && (c == ">=" || c == ">") = if null xs
    then checkGivenIntervalls (createInteriorInterval a b) (createInterval e
    f) else checkIntervalWithMultipleRestrictionsOnDomain ((Cond g a (Var1
    b)) : xs) (Cond h e (Var1 f)) z
88 >        | (a == "<=" || a == "<") && (c == "<=" || c == "<") = if null xs
    then checkGivenIntervalls (createInteriorInterval a b) (createInterval e
    f) else checkIntervalWithMultipleRestrictionsOnDomain ((Cond g a (Var1
    b)) : xs) (Cond h e (Var1 f)) z
89 >        | otherwise = error ("The restrictions made on a domain in state " ++
    show z ++ "are not ok")
90
91 > checkIntervalWithMultipleRestrictionsOnDomain _ _ z = (error "Can not
    compare a variable with a constant in state " ++ show z)
92
93 Here we have the case of multiple restrictions only on the guard of one
    continuous variables using variables as bounds.
94 For the guard it only makes sense to have two restrictions at most.
95 Therefore, we look at the restriction on the domain and see if it can be
    matched against a variable of the guard.
96
97 > checkIntervalWithMultipleRestrictionsOnGuard :: Condition -> [Condition] ->
    Name -> [Char]
98 > checkIntervalWithMultipleRestrictionsOnGuard (Cond _ a (Var2 b)) ((Cond _ c
    (Var2 d)) : (Cond _ e (Var2 f)) : ys) z
99 >        | b == d = compareInterval a c b
100 >       | b == f = compareInterval a e b
101 >       | otherwise = error ("Could not compare " ++ show b ++ " and" ++ show
    d ++ " and " ++ show f ++ " in state " ++ show z)
102
103 Here we have the case of multiple restrictions only on the guard of one
    continuous variables using constants as bounds.
104 For the guard it only makes sense to have two restrictions at most.
105 Therefore, we look at the restriction on the domain and see if it can be
    matched against a constants of the guard.
106
107 > checkIntervalWithMultipleRestrictionsOnGuard (Cond _ a (Var1 b)) ((Cond _ c
    (Var1 d)) : (Cond _ e (Var1 f)) : ys) z
108 >       | a == "==" = []
109 >       | e == c && f == d = checkGivenIntervalls (createInteriorInterval a
    b) (createInterval e f)
110 >       | c == "==" = error ("Multiple restrictions with a == operator in the
    guard in state " ++ show z)
111 >       | e == "==" = error ("Multiple restrictions with a == operator in the
    guard in state " ++ show z)
112 >       | (c == "<=" || c == "<") && (e == ">=" || e == ">") =
    checkGivenIntervalls (createInteriorInterval a b)
    (createIntervalWithTwoBounds e f c d)
113 >       | (c == ">=" || c == ">") && (e == "<=" || e == "<") =
    checkGivenIntervalls (createInteriorInterval a b)
    (createIntervalWithTwoBounds c d e f)
114 >       | otherwise = error ("The restrictions made on a domain in state " ++
    show z ++ "are not ok")
115 > checkIntervalWithMultipleRestrictionsOnGuard _ _ z = (error "Can not
    compare a variable with a constant in state " ++ show z)
116
117 Here we have the case of multiple restrictions on both the domain and the
    guard of one continuous variables using variables as bounds.
```

```
For the guard it only makes sense to have two restrictions at most, however the domain can have more, due to the fact that incoming transitions are considered, as well.
For the domain we use the assumption that the boundaries of the guard from the incoming transition always lays in the boundaries of the domain of the state.
Therefore, we look through every restriction on the domain and see if it can be matched against a variable of the guard.

> checkIntervalWithMultipleRestrictions :: [Condition] -> [Condition] -> Name -> [Char]
> checkIntervalWithMultipleRestrictions ((Cond k a (Var2 b)) : (Cond l c (Var2 d)) : xs) ((Cond i e (Var2 f)) : (Cond j g (Var2 h)) : ys) z
>       | (b == h) && (d == f)  = (compareInterval a g b) ++ (compareInterval c e d)
>       | (b == f) && (d == h)  = (compareInterval a e b) ++ (compareInterval c g d)
>       | b == h = if null xs then compareInterval a g b else checkIntervalWithMultipleRestrictions ((Cond k a (Var2 b)) : xs) ((Cond i e (Var2 f)) : (Cond j g (Var2 h)) : ys) z
>       | b == f = if null xs then compareInterval a e b else checkIntervalWithMultipleRestrictions ((Cond k a (Var2 b)) : xs) ((Cond i e (Var2 f)) : (Cond j g (Var2 h)) : ys) z
>       | d == h = if null xs then compareInterval c g d else checkIntervalWithMultipleRestrictions ((Cond l c (Var2 d)) : xs) ((Cond i e (Var2 f)) : (Cond j g (Var2 h)) : ys) z
>       | d == f = if null xs then compareInterval c e d else checkIntervalWithMultipleRestrictions ((Cond l c (Var2 d)) : xs) ((Cond i e (Var2 f)) : (Cond j g (Var2 h)) : ys) z
>       | null xs = error ("Could not compare " ++ show b ++ " and" ++ show d ++ " and " ++ show f ++ " in state " ++ show z)
>       | (length xs == 1) = checkIntervalWithMultipleRestrictionsOnGuard (head xs) ((Cond i e (Var2 f)) : (Cond j g (Var2 h)) : ys) z
>       | otherwise = checkIntervalWithMultipleRestrictions xs ((Cond i e (Var2 f)) : (Cond j g (Var2 h)) : ys) z

Here we have the case of multiple restrictions on both the domain and the guard of one continuous variables using constants as bounds.
For the guard it only makes sense to have two restrictions at most, however the domain can have more, due to the fact that incoming transitions are considered, as well.
For the domain we use the assumption that the boundaries of the guard from the incoming transition always lays in the boundaries of the domain of the state.
Thus, we first look at the domain, whether it bounds it already from both sides, otherwise we also take the guard into consideration.
If a continuous variable is not bounded by the domain of a state but only in an incoming transition we also take that one into consideration.

> checkIntervalWithMultipleRestrictions ((Cond i a (Var1 b)) : (Cond j c (Var1 d)) : xs) ((Cond k e (Var1 f)) : (Cond l g (Var1 h)) : ys) z
>       | (a == "==" || c == "==") = []

>       | (e == "==" && g == "==") && (f /= h) = error " Can not bound the guard correctly, since a variable is bounded twice with  "=="

>       | (a == "<=" || a == "<") && (c == ">=" || c == ">") && (e == "<=" || e == "<") && (g == ">=" || g == ">") = checkGivenIntervalls
```

```
    (createInteriorIntervalWithTwoBounds d b) (createIntervalWithTwoBounds g
    h e f)
146 >       | (a == ">=" || a == ">") && (c == "<=" || c == "<") && (e == "<=" ||
    e == "<") && (g == ">=" || g == ">") = checkGivenIntervalls
    (createInteriorIntervalWithTwoBounds b d) (createIntervalWithTwoBounds g
    h e f)
147
148 >       | (a == "<=" || a == "<") && (c == ">=" || c == ">") && (e == ">=" ||
    e == ">") && (g == "<=" || g == "<") = checkGivenIntervalls
    (createInteriorIntervalWithTwoBounds d b) (createIntervalWithTwoBounds e
    f g h)
149 >       | (a == ">=" || a == ">") && (c == "<=" || c == "<") && (e == ">=" ||
    e == ">") && (g == "<=" || g == "<") = checkGivenIntervalls
    (createInteriorIntervalWithTwoBounds b d) (createIntervalWithTwoBounds e
    f g h)
150
151
152 >       | (a == ">=" || a == ">") && (c == ">=" || c == ">") = if null xs
    then checkIntervalWithMultipleRestrictionsOnGuard (Cond i a (Var1 b))
    ((Cond k e (Var1 f)) : (Cond l g (Var1 h)) : ys) z else
    checkIntervalWithMultipleRestrictions ((Cond i a (Var1 b)):xs) ((Cond k e
    (Var1 f)) : (Cond l g (Var1 h)) : ys) z
153 >       | (a == "<=" || a == "<") && (c == "<=" || c == "<") = if null xs
    then checkIntervalWithMultipleRestrictionsOnGuard (Cond i a (Var1 b))
    ((Cond k e (Var1 f)) : (Cond l g (Var1 h)) : ys) z else
    checkIntervalWithMultipleRestrictions ((Cond i a (Var1 b)):xs) ((Cond k e
    (Var1 f)) : (Cond l g (Var1 h)) : ys) z
154
155 Due to the assumption of reasonable restrictions.
156
157 >       | (e == "==") = error ("Multiple restrictions with a == operator in
    the guard in state " ++ show z)
158 >       | (g == "==") = error ("Multiple restrictions with a == operator in
    the guard in state " ++ show z)
159
160 >       | otherwise = error ("The restrictions made on a domain in state " ++
    show z ++ "are not ok")
161
162 > checkIntervalWithMultipleRestrictions _ _ z = (error "Can not compare a
    variable with a constant in state " ++ show z)
163
164 The simple case for just one restriction each.
165 We check is the intersection of the case of one restriction each is empty.
166 In case it is not empty we retrieve the overlap.
167
168 > checkInterval :: Condition -> Condition -> Name -> [Char]
169 > checkInterval (Cond a b (Var2 c)) (Cond f d (Var2 e)) z
170 >               | c == e = compareInterval b d c
171 >               | otherwise = error ("Could not compare " ++ show c ++ "
    and" ++ show d ++ " in state " ++ show z)
172 > checkInterval (Cond _ b (Var1 c)) (Cond _ d (Var1 e)) z
173 >               | overlaps (createInteriorInterval b c) (createInterval d e)
    = overlap (createInteriorInterval b c) (createInterval d e)
174 >               | otherwise = []
175 > checkInterval _ _ z = (error "Can not compare a variable with a constant in
    state " ++ show z)
176
```

```
177 This function is called from the functions with multiple restrictions, but
        also could be used in checkInterval.
178 Here we check if an overlap exists, in the case we call our own overlap
        function to calculate the overlap.
179
180 > checkGivenIntervalls x y
181 >   | overlaps x y = overlap x y
182 >   | otherwise = []
183
184 compareInterval compares the intervals, which are bounded by the same
        variables and print out the overlap, if there is any.
185
186 > compareInterval :: [Char] -> [Char] -> Name -> [Char]
187 > compareInterval x y z
188 >           | x == ">=" && y == ">=" = "(" ++ show z ++ ".." ++ show infi
        ++")"
189 >           | x == "<=" && y == "<=" = "(" ++ show ninfi ++ ".." ++ show z
        ++")"
190 >           | x == ">" && y == ">="  = "(" ++ show z ++ ".." ++ show infi
        ++")"
191 >           | x == "<" && y == "<="  = "(" ++ show ninfi ++ ".." ++ show z
        ++")"
192 >           | x == ">=" && y == ">"  = "(" ++ show z ++ ".." ++ show infi
        ++")"
193 >           | x == "<=" && y == "<"  = "(" ++ show ninfi ++ ".." ++ show z
        ++")"
194 >           | otherwise = []
195
196 Just definitions of Infinity.
197
198 > infi = encodeFloat (floatRadix 0 - 1) (snd $ floatRange 0)
199 > ninfi = -(encodeFloat (floatRadix 0 - 1) (snd $ floatRange 0))
200
201 The following functions are used to generate the intervals for bounds with
        constants using the Data.IntervalMap.Interval library.
202 The intervals are generated by comparing comparing the relational operators.
203 Further, we make use of the fact the lower bound is the first argument and
        the upper bound the second argument, in case both bounds exists.
204
205 > createInteriorIntervalWithTwoBounds :: a -> a -> Interval a
206 > createInteriorIntervalWithTwoBounds x y = OpenInterval x y
207
208 > createIntervalWithTwoBounds :: [Char] -> a -> [Char] -> a -> Interval a
209 > createIntervalWithTwoBounds a b c d
210 >       | a == ">" && c == "<"   = OpenInterval b d
211 >       | a == ">" && c == "<="  = IntervalOC b d
212 >       | a == ">=" && c == "<=" = ClosedInterval b d
213 >       | a == ">=" && c == "<"  = IntervalCO b d
214
215 > createInteriorInterval :: [Char] -> Float -> Interval Float
216 > createInteriorInterval x y
217 >                   | x == "==" = OpenInterval y y
218 >                   | x == "<=" = OpenInterval ninfi y
219 >                   | x == ">=" = OpenInterval y infi
220 >                   | x == "<"  = OpenInterval ninfi y
221 >                   | x == ">"  = OpenInterval y infi
222
223 > createInterval :: [Char] -> Float -> Interval Float
```

```
> createInterval x y
>                   | x == "==" = ClosedInterval y y
>                   | x == "<=" = IntervalOC ninfi y
>                   | x == ">=" = IntervalCO y infi
>                   | x == "<"  = OpenInterval ninfi y
>                   | x == ">"  = OpenInterval y infi


Here we print out the overlap of two intervals, which are bounded by constants.
We use the assumption, that we already know that an overlap exists.
Now we have to take the maximum of the two lower bounds and the minimum of the upper bounds, for the new bounds of the overlap.
To see whether the resulting overlap is an open or closed interval, it is crucial to check what kind of interval the bounds for the overlap created.

> overlap :: (Ord a, Show a) => Interval a -> Interval a -> String

> overlap (ClosedInterval lo1 hi1) (ClosedInterval lo2 hi2) = "[" ++ show (max lo1 lo2) ++ "," ++ show (min hi1 hi2) ++ "]"
> overlap (ClosedInterval lo1 hi1) (OpenInterval   lo2 hi2) = (if lo1 <= lo2 then "(" else "[") ++ show (max lo1 lo2) ++ "," ++  show (min hi1 hi2) ++ if hi1 >= hi2 then ")" else "]"
> overlap (ClosedInterval lo1 hi1) (IntervalCO     lo2 hi2) = "[" ++ show (max lo1 lo2) ++ "," ++ show (min hi1 hi2) ++ if hi1 >= hi2 then ")" else "]"
> overlap (ClosedInterval lo1 hi1) (IntervalOC     lo2 hi2) = (if lo1 <= lo2 then "(" else "[") ++ show (max lo1 lo2) ++ "," ++  show (min hi1 hi2) ++ "]"

> overlap (OpenInterval   lo1 hi1) (ClosedInterval lo2 hi2) = (if lo2 <= lo1 then "(" else "[") ++ show (max lo1 lo2) ++ "," ++  show (min hi1 hi2) ++ if hi2 >= hi1 then ")" else "]"
> overlap (OpenInterval   lo1 hi1) (OpenInterval   lo2 hi2) = "(" ++ show (max lo1 lo2) ++ "," ++ show (min hi1 hi2) ++ ")"
> overlap (OpenInterval   lo1 hi1) (IntervalCO     lo2 hi2) = (if lo2 <= lo1 then "(" else "[") ++ show (max lo1 lo2) ++ "," ++ show (min hi1 hi2) ++ ")"
> overlap (OpenInterval   lo1 hi1) (IntervalOC     lo2 hi2) = "(" ++ show (max lo1 lo2) ++ "," ++ show (min hi1 hi2) ++ if hi2 >= hi1 then ")" else "]"

> overlap (IntervalCO     lo1 hi1) (ClosedInterval lo2 hi2) = "[" ++ show (max lo1 lo2) ++ "," ++ show (min hi1 hi2) ++ if hi2 >= hi1 then ")" else "]"
> overlap (IntervalCO     lo1 hi1) (OpenInterval   lo2 hi2) = (if lo1 <= lo2 then "(" else "[") ++ show (max lo1 lo2) ++ "," ++ show (min hi1 hi2) ++ ")"
> overlap (IntervalCO     lo1 hi1) (IntervalCO     lo2 hi2) = "[" ++ show (max lo1 lo2) ++ "," ++ show (min hi1 hi2) ++ ")"
> overlap (IntervalCO     lo1 hi1) (IntervalOC     lo2 hi2) = (if lo1 <= lo2 then "(" else "[") ++ show (max lo1 lo2) ++ "," ++  show (min hi1 hi2) ++ if hi2 >= hi1 then ")" else "]"

> overlap (IntervalOC     lo1 hi1) (ClosedInterval lo2 hi2) = (if lo2 <= lo1 then "(" else "[") ++ show (max lo1 lo2) ++ "," ++  show (min hi1 hi2) ++ "]"
```

```
255 > overlap (IntervalOC     lo1 hi1) (OpenInterval   lo2 hi2) = "(" ++ show
        (max lo1 lo2) ++ "," ++ show (min hi1 hi2) ++ if hi2 >= hi1 then ")" else
        "]"
256 > overlap (IntervalOC     lo1 hi1) (IntervalCO     lo2 hi2) = (if lo2 <= lo1
        then "(" else "[") ++ show (max lo1 lo2) ++ "," ++  show (min hi1 hi2) ++
        if hi1 >= hi2 then ")" else "]"
257 > overlap (IntervalOC     lo1 hi1) (IntervalOC     lo2 hi2) = "(" ++ show
        (max lo1 lo2) ++ "," ++ show (min hi1 hi2) ++ "]"
```

2.6 Printout

```
1  Here we format the output in a readable manner.
2  First, we check if the cycle can show zeno behavior.
3  Afterwards, in case it can show zeno behavior the tool formats the zeno set.
4  In the end, we add a "." to the end of the sentence.
5  getConvergence2 formats the overlapping interval in a reading manner in the
       same way.
6
7  > printZenoSet :: NameCycle -> [(Name, [(Name, [Name])])] -> [Char]
8  > printZenoSet x [] = "The cycle [" ++ unwords x ++ "] does not show zeno
       behavior."
9  > printZenoSet x y  = "The zeno set for the cycle [" ++ unwords x ++ "] is:
       \n" ++ (getZenoSet y)
10
11 > getZenoSet :: [(Name, [(Name, [Name])])] -> [Char]
12 > getZenoSet [] = []
13 > getZenoSet ((a,b):xs) = "In state "" ++ a ++ """ ++ getConvergence b ++
       "\n"++ getZenoSet xs
14
15 > getConvergence :: [(Name, [Name])] -> [Char]
16 > getConvergence [] = []
17 > getConvergence ((a,b):xs) = " for the continuous variable " ++ a ++ " the
       zeno point(s) are: " ++ unwords b ++ ";" ++ getConvergence xs
18
19 > seperateWords [x] = x
20 > seperateWords (x:xs) = x ++ ", " ++ seperateWords xs
21
22 > getConvergence2 :: [(Name, [Name])] -> [Char]
23 > getConvergence2 [] = []
24 > getConvergence2 ((a,b):xs) = "For state "" ++ a ++ """ ++ " the overlapping
       interval(s) are: \n" ++ seperateWords b ++ ";\n" ++ getConvergence2 xs
25
26 > endSentence :: [Char] -> [Char]
27 > endSentence x
28 >       | null x = []
29 >       | otherwise = init (init x) ++ "."
30
31 In the case of NonZenoness we have to check, whether the intersection of the
       boundaries of the domains is empty.
32 In that case the cycle does not show zeno behavior.
33 Otherwise, the zeno set consists of the intersection of the boundaries of the
       domains.
34 Besides that, checkForNonZenoness and checkForZenoness are the same.
```

```
35 First we intersect the interior of the domain with the guard, for each
       transition in the cycle and based on the result we proceed.
36 The result is formatted in a readable manner and returned as the output of
       the functions
37
38 > checkForNonZenoness :: Automata -> NameCycle -> [Char]
39 > checkForNonZenoness x y
40 >                   | null (takeIntersection x y (head $ tail $ reverse y)) =
       case sortBoundaries $ generateBoundaries $ retrieveStateConditions x y
       (head $ tail $ reverse y) of
41 >             [x] -> endSentence $ printZenoSet y [x]
42 >             x   -> endSentence $ printZenoSet y $ intersectBoundaries x
43 >                   | otherwise = "The overlapping  intervals for the cycle
       [" ++ unwords y ++ "] are: \n" ++ endSentence (getConvergence2
       (takeIntersection x y (head $ tail $ reverse y))) ++ "\nThus, no clear
       statement about zenoness can be made."
44
45 > checkForZenoness :: Automata -> NameCycle -> [Char]
46 > checkForZenoness x y
47 >                   | null (takeIntersection x y (head $ tail $ reverse y)) =
       endSentence $ printZenoSet y $ sortBoundaries $ generateBoundaries $
       retrieveStateConditions x y (head $ tail $ reverse y)
48 >                   | otherwise = "The overlapping intervals for the cycle ["
       ++ unwords y ++ "] are: \n" ++ endSentence (getConvergence2
       (takeIntersection x y (head $ tail $ reverse y))) ++ "\nThus, no clear
       statement about zenoness can be made."
```