U N I K A S S E L
V E R S I T Ä T
Faculty of Electrical Engineering and Computer Science
Reseach Group Theoretical Computer Science/Formal Methods

SoPHY - A specification language for hybrid systems

A thesis submitted to the
Faculty of Electrical Engineering and Computer Science
of the University of Kassel
in partial fulfillment of the requirements for the degree of

Master of Science

Submitted from:	Lara Yörük Mi
Matriculation Number:	30219418
Department:	Fachgebiet Theoretische Informatik/Formale Methoden
Examiner:	Prof. Dr. Martin Lange Prof. Dr. Jörg Kreiker (Hochschule Fulda)
Supervisor:	Dr. Norbert Hundeshagen
Submitted:	06.09.2017

Abstract

This work contributes to the three subjects, hybrid systems, modeling languages and educational software. We develop the specification language for hybrid systems, SoPHY. The aim of this language is to describe scientific experiments. These experiments are modeled as hybrid automata. On the basis of SoPHY, a simulation tool can be developed, which is oriented towards didactic use in the context of *self-regulated learning.*
The specification language is easily readable and self-explanatory by its close reference to experiments. It is also easy to process by computers since its design is adapted to the well-known JSON format. This facilitates its later use in the software.

Zusammenfassung

Diese Arbeit leistet einen Beitrag zu den drei Themen, hybride Systeme, Modellierungssprachen und Bildungssoftware. Hierbei wird die Spezifikationssprache für hybride Systeme, SoPHY, entwickelt. Das Ziel dieser Sprache ist, wissenschaftliche Experimente zu beschreiben. Diese Experimente werden in Form von hybriden Automaten modelliert. Auf Basis von SoPHY kann dann ein Simulationstool entwickelt werden, welches auf den didaktischen Einsatz im Themenbereich *Selbstgesteuertes Lernen* ausgerichtet ist.
Die Spezifikationssprache ist durch ihre nahe Anlehnung an Experimente leicht lesbar und selbsterklärend. Sie ist außerdem leicht durch Computer zu verarbeiten, da ihr Aufbau an das bekannte JSON Format angepasst ist. Das erleichtert ihre spätere Verwendung in der Software.

Declaration

Herewith I declare, that I have made the presented paper myself and solely with the aid of the means permitted by the examination regulations of the University of Kassel. The literature used is indicated in the bibliography. I have indicated literally or correspondingly assumed contents as such.

Kassel, 06.09.2017

Lara Yörük

Contents

List of Figures

Listings

1 Introduction

1.1 Motivation

Nowadays most children grow up using computers in their daily life. As a logical consequence the advantages of computers are also recognized in the didactical scope. Thus, technology in education plays a more and more important role in research and practice, which can be seen by the fact that the german *Federal Ministry of Education and Research* strongly supports projects in this area [Bun]. Computers have the power to solve a lot of problems from various topics. For instance, they can be used as calculators for mathematical problems or –with an internet connection– for online researches and to provide teaching materials on online platforms. But they can also be used to entirely replace single teaching units. The Application DiVoX, which is shown later, serves as an example that also indicates that computers are a vital component in the context of self-regulated learning.

Definition 1.1 *Self-regulated learning [GNH13]*
Self-regulated learning is a form of acquiring knowledge and skills in which the learners are independent and self-motivated. Learners independently choose their own goals and learning strategies that will lead to achieving those goals. It is through evaluating the effectiveness of one's learning strategies — comparing one's current state with the target state — that learning can be modified and optimized.

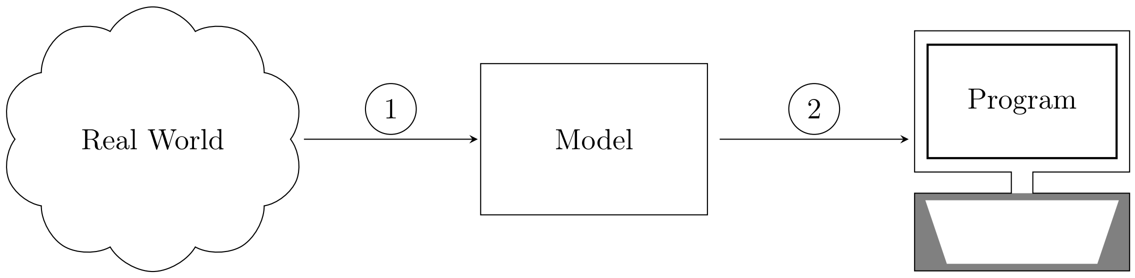
While a lot of methods on the subject of educational technology, like interactive whiteboards, computer algebra systems and systems to manage electronic resources, were yet mainly used to support the teacher, software in the context of self-regulated learning can result in a major change. By applications that are developed in this scope, the way of teaching can really be changed. On that account some examples can be found in Section 1.3. Those tools are used by students to learn without the need of a teacher performing the classical direct instruction. The students can find their own speed which suits them the best, define their own goals and thus, they motivate themselves to learn. The aim of this work is to make a contribution in the context of self-regulated learning by working towards a tool that will be able to simulate scientific experiments.

The tablet application DiVoX [HMP+] is the basis for this work. It is used by the Didactics of Biology department at the University of Kassel to examine the use of digital media in teaching. By means of the app teachers can create a whole teaching unit on the subject of a scientific experiment with a tablet. Every experiment is segmented in multiple steps, like hypothesis, planning, execution and result. For each of these steps teachers can add work materials in form of texts, images, videos or multiple-choice questions. After a teacher completed a teaching unit he can divide the students of his class into groups and assign them to this unit. While performing the teaching unit, each group of students gets a tablet and they can log in with their personal data. The materials that were added by the teachers are presented to the students step by step and they can add their answers, again, in form of texts, images or videos. Even if problems occur, the students do not necessarily have to consult the teacher, because he can add multilevel help information to every step. Thus the students can look at these information when they start to have understanding problems. In the execution step the students really have to perform the experiment and they can protocol their results in the app. Finally after the teaching unit is finished, the teacher is able to look at the given answers of each group, on his tablet and to evaluate the unit in this way. The application DiVoX is a good example for a tool on the subject of self-regulated learning because the students can determine the progress of the teaching unit by theirselves while they are supported by the app. They can work with their own speed and acknowledge help information if they want to. This application is actually the starting point of this work, since with DiVoX it is still necessary that the students perform the experiments in reality. To do so they certainly have to be in school, because they need the tools to build the experiment. Moreover in the majority of cases there is only one chance to perform the experiment. Thereby the students can not estimate possible effects that would be caused by changing the process. On these grounds the aim of this work is to create a possibility how scientific experiments can be simulated by a computer.

When something from the real world should be brought to the computer, like scientific experiments, the subject needs to be formulated in a computer understandable way. This process is called *modeling* and is explained in Section 2.1. When experimenting, students create hypotheses and want to test, whether they are valid. Scientific experiments have discrete and continuous properties, which necessarily have to be modeled. A model which meets this requirements is a hybrid system. Hybrid systems unite discrete and continuous behavior, where the continuous parts evolve over time and discrete events can be modeled by adding conditions that determine when these events happen. More information on hybrid systems can be found in Chapter 2. In this section we will also see that hybrid systems are a good way to model scientific experiments, because they can handle the flow of the properties over time with the continuous parts and they are able to define the occurrence of special events with the discrete behavior.

Therefore the decision was made to develop SoPHY, as a specification language for modeling scientific experiment with hybrid systems, and the simulation tool that is based on SoPHY. The tool can be used in the context of self-regulated learning, like it was mentioned at the beginning of this section. Teachers will be able to model experiments with

Figure 1.1: Modeling



the tool and students can fill them with parameters to determine if they can corroborate their hypothesis. Thus, the students can not only simulate the experiments at home without the materials they normally need in school, they can also perform the experiments in multiple ways by changing the parameters, in a time in which they could never do it in the conventional way.

In Figure 1.1 we can see the process that needs to be performed to develop the mentioned tool. The task of this thesis is to model experiments via hybrid systems (1). The second step is (2), to use this model in a tool for simulation, developed in another thesis by Orcun Yörük [Yör17], parallel to this one.

1.2 Task and Outline

The task for this master thesis is to develop a language to describe hybrid systems. The aim of the language is, to be used by a tool, which is applied in the context of self-regulated learning for the simulation of scientific experiments. Since hybrid systems are used for modeling the experiments, the topic of verification is interesting in addition to simulation. This can be used to prove the correctness of a system or to check certain theses. For example it could be checked whether in any possible configuration of the experiment the population of a species has doubled after a certain time. With simulation the answer to such questions is not always clear. The big difference is that a simulation is only a particular run of the experiment. Many experiments are not deterministic, in the sense that they are probabilistic. This means that at certain points in the experiment there is more than one possible further course. In the simulation, one of these courses must be selected and it remains unclear what would have happened in the others. In verification, the entire system is viewed, and thus global specifications such as those mentioned above can be checked. Therefore the thesis shall also give a possibility how to verify the hybrid systems that are modeled within the language. This work is structured as follows:

In Chapter 2 an overview about hybrid systems is given in Section 2.3. In Section 2.4 their representation as automata is discussed. Regarding the aim to simulate these systems in Section 2.5.2 a calculation of an execution is shown.

Next, in Chapter 3 we focus on the design process of the language SoPHY. The syntax for SoPHY is defined in Section 3.2. The defined syntax is intended to be used in didactics,

which is why it is kept simple. Regarding the defined syntax in Section 3.3 it is revealed how a system, which was modeled with SoPHY, can be translated into a hybrid automaton, the standart specification model for hybrid systems.
In Chapter 4 the reader gets a closer look on how SoPHY is used. Therefore a set of example experiments is described and it is shown how these can be modeled with SoPHY as hybrid systems.
Chapter 5 explains how a specification in SoPHY can be translated to HSolver, a tool to verify hybrid systems. We will also see how the verification fits into the application of this thesis.
The conclusion and an outlook on the future use of SoPHY can be found in Chapter 6.

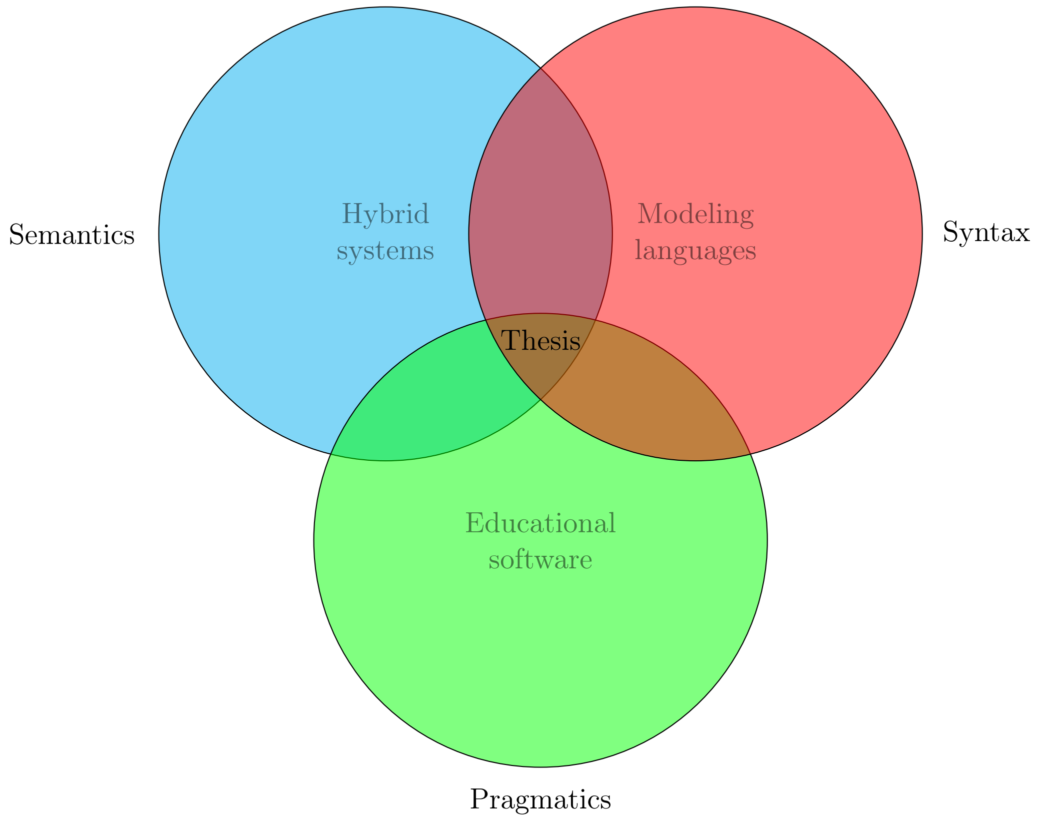
1.3 Related Work

This thesis is a combination of three main topics: hybrid systems, modeling languages and educational software. In Figure 1.2 this coherence is illustrated. The three subjects are labeled with the words semantics, syntax and pragmatics. That is because the model of hybrid systems is used as the semantics of the experiments. SoPHY, the language to model the experiments forms the syntax in which the experiments are formulated. And finally educational software is the subject where SoPHY will be applied, that is why it is denoted as the pragmatic part of this thesis.
Before presenting SoPHY, this section will briefly review some of the work in these three subjects.

1.3.1 Work on hybrid systems

Hybrid systems were introduced by Thomas A. Henzinger in 1996 [Hen96]. He stated a formal definition for hybrid systems and classified them "according to what questions about their behavior can be answered algorithmically". In the same time he also published an article about the algorithmic analysis of hybrid systems [ACH+95]. To the best of my knowledge, there are currently no publications about hybrid systems that are used in the terms of didactics available, so this thesis will be a first step in this particular field. Nevertheless there are several languages to model hybrid systems, which should be mentioned. In 2000 the language CHARON [AGH+00] was introduced. With CHARON interacting hybrid systems could be specified modularly. CHARON is used for simulation and verification of hybrid systems. Some other tools that are used for simulation are Simulink [ABRW17], an industrial tool, HyVisual [BCL+05] and SHIFT [DGS00] which are developed in academia. Simulink is integrated into the popular environment MATLAB and is not only made for hybrid systems. Therefore the language is more difficult to use than the languages that are specially used in the context of hybrid systems. An advantage

Figure 1.2: Subjects of this thesis



is that it is a graphical language whose syntax is very intuitive. On the academic side, *The Hybrid System Visual Modeler* HyVisual also uses a graphical language to model the system and it is, like the name already says, especially made for hybrid systems. The model that is created with the graphical language is saved in he popular XML format. As graphical languages are more useful for simpler hybrid systems, SHIFT is a language that can be used for highly complex hybrid systems. SHIFT, which is a permutation of the acronym of *Hybrid Systems Tool Interchange Format*, can describe networks of hybrid automata, whose components can be created, interconnected and destroyed during the evolution of the system.

1.3.2 Work on modeling languages

A pivotal moment for the work on modeling languages for dynamical systems was the introduction of CCS [Mil80], short for *Calculus of Communicating Systems*. With CCS the interaction of processes could be modeled. In 1990 a calculus that extends CCS with probability and time [HJ90] was published. With this calculus it is possible to model real-time aspects of distributed systems. Another process calculus that arose in the same time as CCS is CSP, *Communicating sequential processes* [BHR84]. CSP is very similar to CCS which originates from the fact that both works were influenced by each other. As a proceeding of his work on CCS the π-calculus was developed in 1992 [MPW92]. The

π-calculus is very expressive and can, in contrast to CCS, model processes, which have changing structure. After that a lot of modeling languages were, where the most important for this paper was the introduction of *timed automata* in 1994 [AD94]. Timed Automata are an interesting subclass of hybrid systems where all continuous variables are real timed clocks and the conditions and resets are more limited. They are interesting because the reachability problem for timed automata is, unlike for hybrid automata, decidable. In 1997 a modeling language for timed automata, UPPAAL, was published [LPY97]. UPPAAL is used for simulating, specifying and verifying safety and bounded liveness properties for real-time systems that are modeled as networks of timed automata.

1.3.3 Work on educational software

One example of tools for self-regulated learning is GeoGebra. It is a common education tool for students, which motivates the students to work independently, because the students simply start to test what they can do with mathematics. GeoGebra connects geometry, algebra, tables, mathematical drawings, statistics and analysis in one software [Int]. The work on GeoGebra started in 2002 [Hoh02]. The aim was to unite the features of computer algebra systems, which handle geometrical objects by their equations, and dynamical geometry software, which can be used to draw geometrical object on the computer. The students should be able to change the equation of an object and this should result in changing the geometrical figure. In the same way, by moving the figure, the equation will be changed dynamically. Thus, the students can learn about this coherence by themselves. The work on GeoGebra was continued in 2006 [Hoh06] and it is until now proceeded by the International GeoGebra Institute. Recently, GeoGebra has been enhanced with Theorem-Proving techniques. This extends the usage of GeoGebra in terms of self-regulated learning once more.
Another set of tools which also interact with the student were published between 2002 and 2004 by the University of Hildesheim on the subject of fractions [Hen]. The tools are called *Mathematik heute - Bruchrechnung, Maßstab - Bruchrechnung* and *Welt der Zahl - Bruchrechnung.* All tools have the same use case, but are adapted to three different textbooks. With this tools the students can learn to calculate with fractions by going through all the steps of the calculation. These tools are also able to recognize mistakes that are made by the students and give hints what they did wrong. For example when a student should solve the calculation $\frac{5}{3} + \frac{2}{5}$ and answers with $\frac{7}{8}$, the program gives the hint that before adding the fractions they need to be converted to a common denominator.

2 Preliminaries

At the beginning of this section we will examine how scientific experiments can be modeled. First we will look closely on the terms of modeling and then we will introduce transition systems by determining the characteristic parts of experiments. After that, there is a formal definition of hybrid systems that is on a very general level. A hybrid system is a system with discrete and continuous behavior. The discrete behavior of the system corresponds to changes of states in a finite automaton. The continuous behavior is modeled by real valued variables that evolve over time. Afterwards in Section 2.4 we will define the concept of hybrid automata, where parts of the definition are more limited, for the reason how a hybrid system is represented. For the subject of this paper we do not need to work on this highly generic level. We will use the limited representation, hybrid automata, because their graphical representation is easy to understand and totally fits our needs.

2.1 Modeling experiments

To simulate scientific experiments with a computer, these experiments need to be modeled so that they are machine-readable. But what does it mean to model experiments? In Figure 1.1 the concept of modeling is shown. Modeling is the process of creating a model that abstracts a part of the real world (1), so that the computer can perform calculations or work in other ways with this model (2). In this thesis the model needs to be a formal representation that uses methods from mathematics or computer science because the model should be used by a tool for simulation. As mentioned in the last section we use hybrid systems to define a model for a scientific experiment.
As a first step, we need to determine in which way scientific experiments could be modeled. Thus, we need to look at the properties that characterize them. While the students hypothesize, they are mostly talking about values of some properties of the experiment. We now take a look at the following experiment.

Example 2.1 *Water flea experiment*
We consider a simple experiment with the aim to examine the growth of a water flea population. The setup of the experiment consists of a container which is filled with water. The water fleas are located inside this container and are fed regularly. The students should examine how fast the population of the water fleas grows, assuming that the outside conditions, like temperature and light, remain constant. Thus, the students need to measure the amount of water fleas periodically.

The central question for the experiment in Example 2.1 is "How many water fleas are inside the container after a certain time?". The important properties for this experiment therefore are time and the amount of water fleas. Such a pattern can also be found whilst considering other experiments. In case of physical experiments, for example ones with a pendulum, aspects like velocity or the angle of the pendulum are examined. In experiments in physics, objects are always placed in situations in which they are exposed to physical laws. In order to examine these, the state of the object is viewed at certain times. It is also possible to recognize such a procedure in biological experiments. This leads to the assumption that the important parts for a certain class of experiments are time and additional properties that are interesting for the hypothesis. When the students build the setup for the experiments, they implicitly define a dynamic interaction between various objects that take part in the experiment, for example an organism and light. To represent such circumstances, we need a dynamical system that on the one hand can handle the flow of some properties over time and on the other hand it should also provide the ability to define moments when something changes discretely. This is the reason why we decided to use hybrid systems.
Before we can begin to model experiments, we need to explain modeling. Therefore we again use the experiment of Example 2.1. First of all, it should be pointed out that the real world is not exactly depicted while modeling. It is only modeled what is really important. A model abstracts from unimportant details, that are not relevant for the current task. In the water flea experiment, for example, the gender distribution is not relevant for the study of the growth of the fleas. Therefore, the gender is completely ignored in the model. In reality, for example, a student might overthrow the container in which the water fleas are located. For simulations this kind of information is absolutely not important and thus, such circumstances are omitted in the model. A model can also contain simplifications. This means that influences from unknown data are ignored or simplified. In reality, of course, it is not possible to permanently create exactly the same conditions. The temperature of the water is affected for example by the ambient temperature, etc. However, this fact is not of great importance, therefore, it is considered simplified that e.g. the temperature is always the same and optimal conditions for growth are created. Moreover complicated effects, for example the disease of a water flea or its infertility can also be omitted or used simplified [EGK11], by assuming that every 1000th flea is sick or barren.

2.2 Transition Systems

Transition systems are a common model in computer science to represent dynamical systems. They describe the possible states of a systems and how the system can change between these states. A transition system can be seen as a directed graph, whose nodes represent the states of the system and whose edges model the state changes, called transitions [BK08].

Definition 2.1 *Transition System [BK08]*
A transition system T is a tuple $(S, S_0, Lab, \rightarrow)$ where

- *S is a set of states,*
- *$S_0 \subseteq S$ is a set of initial states,*
- *Lab is a set of action labels,*
- *$\rightarrow \subseteq S \times Lab \times S$ is a transition relation.*

In the following we will see an experiment that we will take as an example several times in the course of the work. In this section we will examine it more closely and build a transition system from it.

Example 2.2 *Water tank experiment*
As an example for a simple experiment consider a water tank whose water level should be kept between a lower bound and an upper bound. As illustrated in Figure 2.1, the tank has a valve which is used to connect a pump that fills in the water. The illustration also shows, that the water tank has a small leak at the bottom, where water constantly drops out. The red lines mark the boundaries between which the water level should be kept. During the experiment, the procedure is the iteration of the following steps:

1. *Pump water into the tank until the maximum height is reached. While the pump is turned on, the evolution of the height can be described by the ordinary differential equation (ODE)*
$$\dot{h} = 1 + (-c \cdot \sqrt{h}),$$
where c is
$$c = -\frac{\textit{Leak size}}{\textit{Tank size}}\sqrt{\frac{2 \cdot g}{1 - \frac{\textit{Leak size}}{\textit{Tank size}}^2}}$$
and g is the gravity acceleration ($981 cm/s^2$).

2. *Turn off the pump until the minimum height is reached (due to the loss of water by the leak). Here the ODE for the height is:*

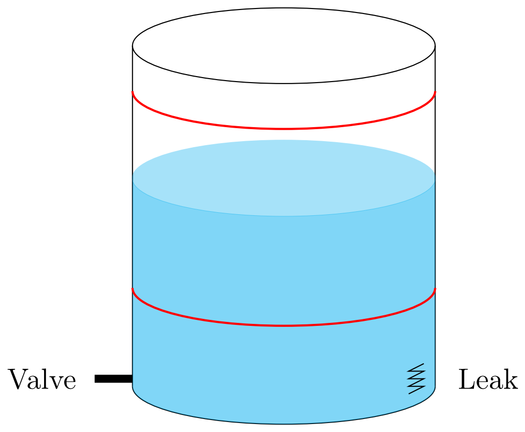
$$\dot{h} = -c \cdot \sqrt{h}$$

and c is the same as before.

3. *Open the valve and connect the pump. While the valve is opened, water will pour out until the pump is connected, which takes 3 seconds. The evolution of h is described with the same ODE as if the pump is turned off. But the size of the leak is now the size of the real leak plus the size of the valve.*

This water tank is based on an example from [Pla10].

Figure 2.1: Water Tank



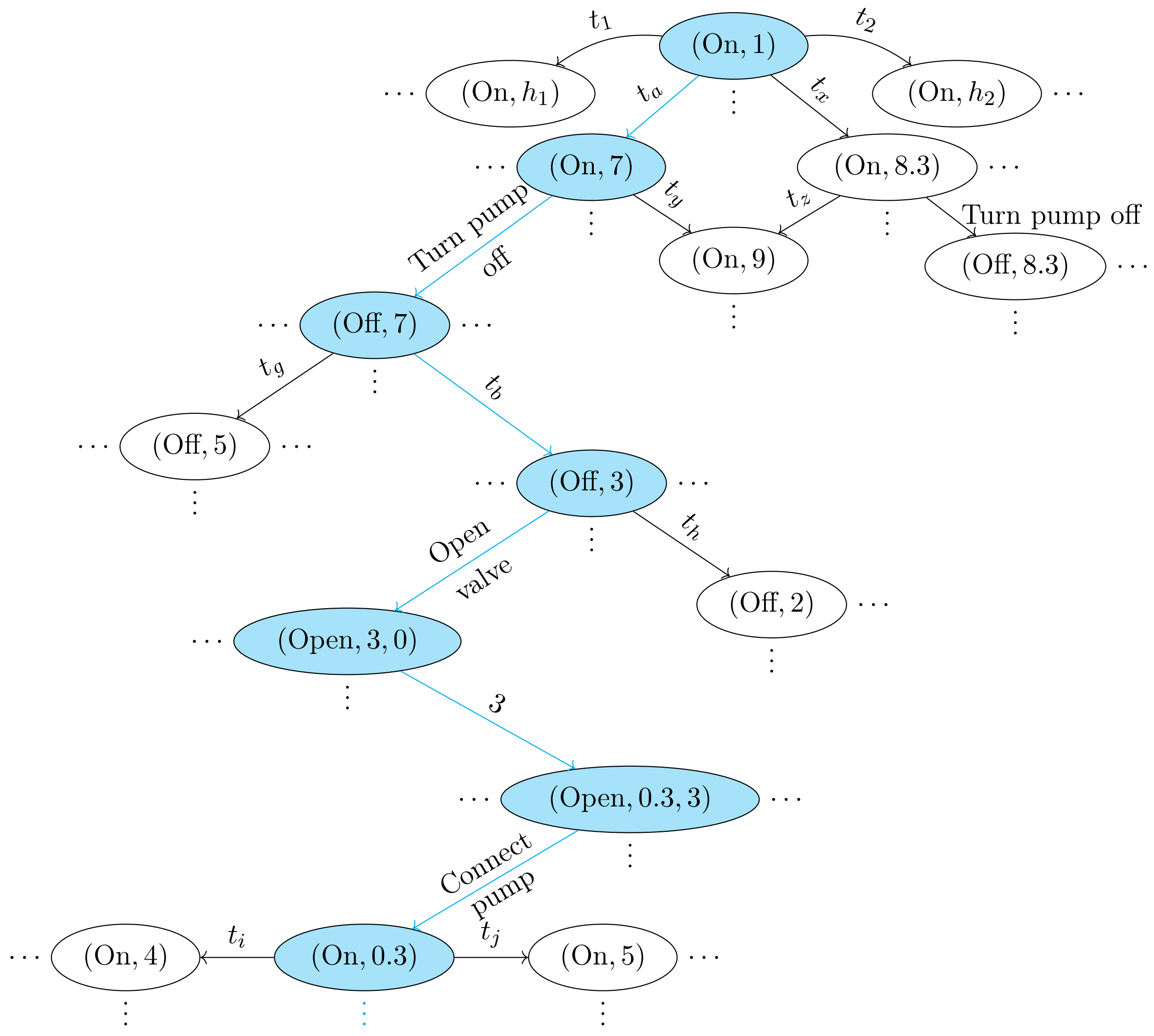
Using the experiment from Example 2.2 we will now find out which properties are the most important, which can not be omitted during the modeling. Therefore, we examine the experiment on characteristics that are used to describe the current state. It is obvious that it is pointless to talk about the state of an experiment without specifying a time at which this state occurs. Thus, an important feature is the time. If we now look at the water tank experiment at a certain point in time and want to describe it, it is important to mention how high the water level in the tank is. Values such as the temperature or the pH value of the water are of no interest to our experiment, but in others it of course might be. In addition to the water level, it is also important to know in which position the pump is located. To summarize, we now know that we can describe the state of our experiment at a given time by the water level and the state of the pump. The next step is to figure out how these values, i.e. the water level and the pump, change over the course of the experiment. The description of the experiment clearly defines when the level of the pump changes. The pump is switched off as soon as the maximum level is reached. The valve is opened as soon as the water level drops to the minimum and after 3 seconds the pump is switched on. The status of the pump is therefore always linked to certain conditions.

Finally it is to be defined how the water level changes. The change follows the specified ODEs, which however may be different depending on the state of the pump. In order to find out the water level, the level of the pump must also be known. Then the respective ODE can be used to calculate the water level.
Now we have worked out the characteristics of the experiment, which are indispensable. We now want to create a transition system that describes the possible sequences of the experiment. As defined in Definition 2.1, a transition system is a tuple $T_{WT} = (S, S_0, Lab, \rightarrow)$. We know that the state of the system can be described by the water level and the state of the pump. For our transition system, this means that the set states S is equivalent to $S = \{\text{On, Off, Open}\} \times \text{water level}$. Next, we need to define the initial set of states S_0. This depends on how the experiment is set up at the beginning. We assume that at the beginning of the experiment the pump is always on and the water level is at the minimum. Therefore, the set of initial states is $S_0 = \{(\text{On}, 1)\}$. The set Lab contains labels that describe the actions, which lead to a change of the state of the system. In this example the state can change in two ways: Either the status of the pump/valve changes or just the water level changes. On the one hand, the water level changes when time evolves, so we need all possible times as labels. On the other hand we need some labels that describe the changes of the pump/valve. Finally we get $Lab = \mathbb{R} \cup \{\text{Turn pump off}, \text{Open valve}, \text{Connect pump}\}$. The last element of the tuple is the transition relation $\rightarrow$. To figure out which transitions can be taken, we start at the initial state and determine which states can be reached. In the following h' denotes the value that is calculated with the ODEs from Example 2.2 for time t. Therefore we get the transitions $\rightarrow_1 = \{((\text{On}, h), t, (\text{On}, h'))\}$ for the transitions that can be taken when only the height changes while time t has evolved. The transitions that can be taken to change from *On* to *Off* are $\rightarrow_2 = \{((\text{On}, h), \text{Turn pump off}, (\text{Off}, h)) \mid h \geq \text{maximum height}\}$. Now there are again some transitions where only the height changes and the pump stays off: $\rightarrow_3 = \{((\text{Off}, h), t, (\text{Off}, h'))\}$. For the transitions where the status changes from *Off* to *Open* there are the transitions $\rightarrow_4 = \{((\text{Off}, h), \text{Open valve}, (\text{Open}, h)) \mid h \leq \text{minimum height}\}$. While the valve is open there are again the transitions where the status does not change, but the connection of the pump will need $3s$: $\rightarrow_5 = \{((\text{Open}, h), t, (\text{Open}, h')) \mid t < 3\}$. The last set of transitions are the ones when the status changes from *Open* to *On*. To assure that the valve was open for $3s$, we need to introduce a variable x that counts the time. x will also be existent in all the other states, but is not needed, so we omit it there. When the valve is open, x evolves with $\dot{x} = 1$, just like a clock. Accordingly, the last set of transitions is $\rightarrow_6 = \{((\text{Open}, h, x), \text{Connect pump}, (\text{On}, h)) \mid x = 3\}$. Taken together, the transition relation $\rightarrow$ of the system T_{WT} is $\rightarrow\, = \,\rightarrow_1 \cup \rightarrow_2 \cup \rightarrow_3 \cup \rightarrow_4 \cup \rightarrow_5 \cup \rightarrow_6$.
To visualize transition systems, they can be seen as directed graphs, where a path of the graph is a run of the system over time. The graph $G_{T_{WT}}$ for the transition system T_{WT} is partially shown in Figure 2.2, where we set the maximum height to $7cm$ and the minimum height to $3cm$. The path consisting of the colored vertices and edges, is representing the beginning of a run that fills the tank until the water level is $7cm$. Afterwards the water level falls to $3cm$ and the valve is opened for $3s$ before the pump is filling the tank with

water again. It is obvious, that the pictured transition system is infinite (the dots indicate infinite other transitions).

Now that we have worked out the most important features of an experiment and have seen how to describe it through a transition system, we return to hybrid systems. Hybrid systems give exactly the capabilities we need to capture all the important information of an experiment. Furthermore, their denotational semantics accurately describes such a transition system as we have created it from the information.

Figure 2.2: Water Tank Transition System



2.3 Hybrid Systems

Now we want to focus on hybrid systems to model the experiments. In Section 2.5.2 we will see, that the semantic of hybrid systems can be described by transition systems.
This Section 2.3 is mainly based on [Hen96], [LTS08] and [ACH$^+$95].

The hybrid system H is a collection $H = (L, X, Lab, E, Act, Inv)$ where the following is defined:

- The finite set L denotes the discrete values . The elements $l \in L$ are called locations.
- The finite set X denotes the continuous variables.
- The function $Inv\colon L \to 2^{\mathbb{R}^{|X|}}$ assigns an invariant to each location $l \in L$.
- The function $Act\colon L \to (\mathbb{R}^{|X|} \to (\mathbb{R} \to \mathbb{R}^{|X|}))$ denotes the continuous changes in the variables $x \in X$. This function assigns a function $f\colon \mathbb{R}^{|X|} \to (\mathbb{R} \to \mathbb{R}^{|X|})$ to each location $l \in L$, which is called the activity of the location l.
- The finite set Lab annotates the transitions. The elements $a \in Lab$ are called labels.
- The set $E \subseteq L \times Lab \times 2^{\mathbb{R}^{|X|}} \times \mathbb{R}^{|X|} \times L$ denotes the changes between the discrete values. We call the elements $e \in E$ transitions. Each transition $e = (l, a, G, r, l')$ consists of the source location $l \in L$, the target location $l' \in L$, the label $a \in Lab$, the guard $G \in 2^{\mathbb{R}^{|X|}}$ and the reset $r \in \mathbb{R}^{|X|}$.

2.4 Hybrid Automata

In Section 2.3, the formal definition of a hybrid system was introduced. In this section, we look at hybrid automata. In the literature on hybrid systems, the use of hybrid automata is confused. Often they are also used synonymously. In this work, we want to clearly delineate hybrid automata from hybrid systems.
Hybrid systems are a general description of dynamic systems. We introduce hybrid automata as a syntactic representation of a special class of hybrid systems. A hybrid automaton is depicted by a directed graph where the set of vertices is finite and the vertices are labeled with the invariants and activities. We will now define this special class of hybrid systems, called hybrid automata, by considering each element of the tuple $H = (L, X, V, Lab, E, Act, Inv)$.

The locations $l \in L$ form the vertices of the automaton.
Each vertex contains its invariant $Inv(l)$. We restrict the invariants to conditions, $\langle Cond\rangle$, that are described by the following intervals, according to the grammar

$$\langle Cond\rangle \ ::= \ \langle Term\rangle\ \alpha\ \langle Term\rangle \mid \langle Cond\rangle\ \beta\ \langle Cond\rangle \mid \neg\ \langle Cond\rangle$$

$$\langle Term\rangle \ ::= \ \langle Term\rangle\ \gamma\ \langle Term\rangle \mid \mathrm{c} \mid x$$

where $\alpha \in \{=, <, >, \leq, \geq\}$, $\beta \in \{\wedge, \vee\}$, $\gamma \in \{+, *, -\}$, $x \in X$ and $c \in \mathbb{R}$.
The vertices also contain the activity $Act(l)$ of the corresponding location l. For hybrid automata, we restrict the form of the activity. A function of the form $f : \mathbb{R}^{|X|} \to (\mathbb{R} \to \mathbb{R}^{|X|})$ gives a function $g : \mathbb{R} \to \mathbb{R}^{|X|}$ for an element $\tilde{x} \in \mathbb{R}^{|X|}$. In order to describe the activities in a hybrid automata, we restrict these functions to those that can be describes by systems of first order ODEs. The element $\tilde{x} \in \mathbb{R}^{|X|}$ describes the initial values with which the system of ODEs can be solved. The solution is the mentioned function g. By using a system of ODEs with initial values to describe the activity, there is a unique solution (see Section 2.6.2), what is necessary for simulation. Obviously this is a strong limitation, because with the previous definition much more could be represented. For example, even partial or unsolvable differential equations could be produced there. In the following, we will see some examples from [Sta16] to justify that ODEs are sufficient for a variety of experiments. For example the ODE

$$\dot{x}(t) = kx(t)$$

denotes the simple exponential growth, which is used to model the malthusian population growth. Also the non-malthusian population growth, which considers the decrease of the population due to a competition for resources, can be modeled by the first order ODE

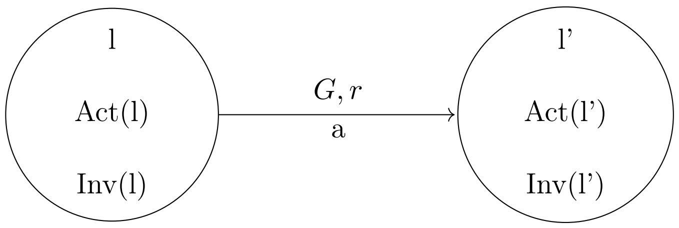
$$\dot{x} = rx(1 - \frac{x}{k}).$$

Even enzymatic reactions can be modeled by first order ODEs, for example with the Michaelis-Menten equation or the Hill equation from [Sta16].
But not only biological applications can be modeled with first order ODEs. In [Gre12] there are a lot more examples, like the radioactive decay of carbon-14, plutonium-241, radium-226 and thorium-234 with

$$\dot{m}(t) = -km(t).$$

And also differential equations for electric circuits, like the ones for resistors, inductors and capacitors can be modeled with first order ODEs.
Even if first order ODEs are not enough to describe an experiment, every higher order ODE can be transformed to a system of first order ODEs [Col81]. Since we use systems of first order ODEs, we can also model behavior that is described by higher order ODEs.

Figure 2.3: Edge of a Hybrid Automaton



We have seen that the nodes of the automaton represent the locations $l \in L$, which in turn contain the activities *Act* and the invariants *Inv*. Now we consider the elements X, *Lab* and E. The sets X and *Lab* result directly from the hybrid system. For each transition $e = (l, a, G, r, l') \in E$ the hybrid automaton has an edge from the vertex l to the vertex l' labeled with a, like it is depicted in Figure 2.3. Every edge contains the guard G that has the same form as the invariant of a vertex and the reset r which is given as a list of value assignments for each continuous variable $x \in X$ of the form

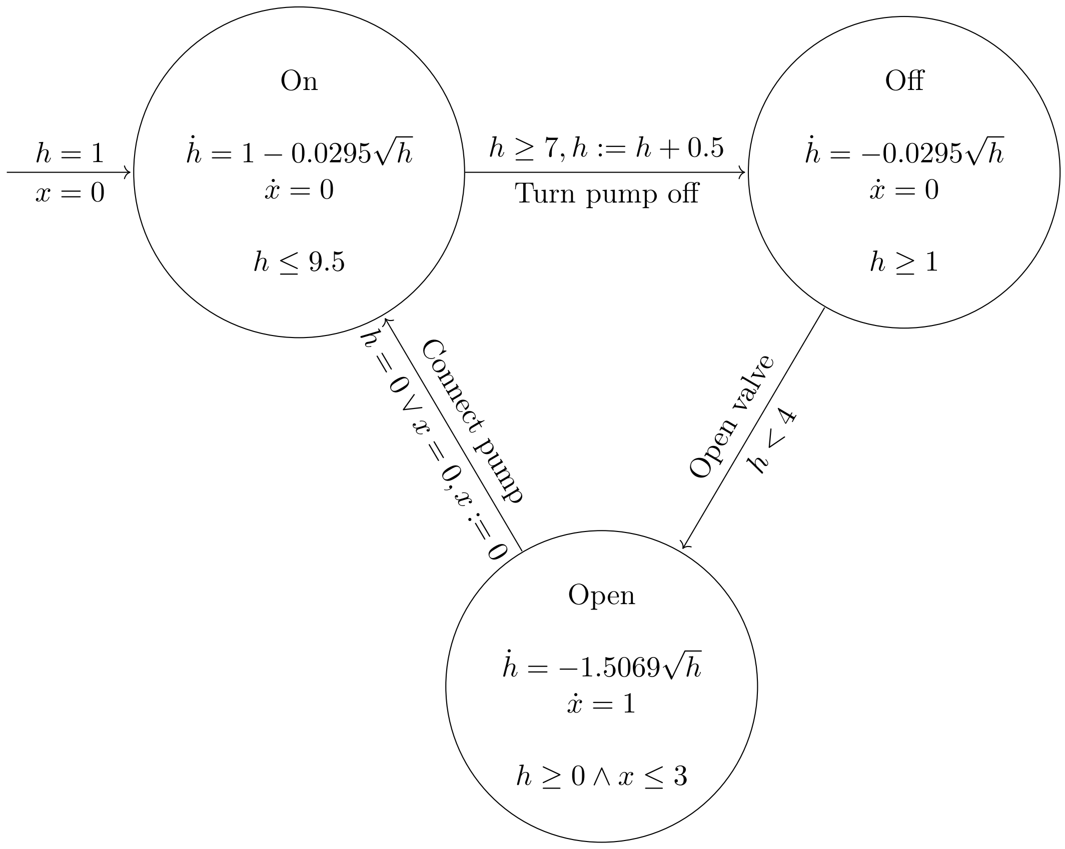
$$\langle \mathit{Reset} \rangle \ ::= \ x := \langle \mathit{Term} \rangle$$

where $\langle Term \rangle$ is formed like we have seen for the invariant. It means that the value of x after transition e is the solution of the expression $\langle Term \rangle$. If there is no reset for some $x \in X$ on an edge we assume that the reset is $x := x$ and the value of the continuous variable x does not change during the discrete transition.

Example 2.3 *As an example, the hybrid automaton $H_{\text{WaterTank}} = (L, X, Lab, E, Act, Inv)$ for the water tank from Example 2.2 is shown in Figure 2.4, where the following is defined:*

- $L := \{\mathit{On}, \mathit{Off}, \mathit{Open}\}$,
- $X := \{h, x\}$, *where h is the height and x is a clock for connecting the pump,*
- $\mathit{Lab} := \{\mathit{Turn\ pump\ off},\ \mathit{Open\ valve},\ \mathit{Connect\ pump}\}$,
- $E := \{(\mathit{On}, \mathit{Turn\ pump\ off}, G_1, r_1, \mathit{Off}),\ (\mathit{Off}, \mathit{Open\ valve}, G_2, r_2, \mathit{Open}),$
  $(\mathit{Open}, \mathit{Connect\ pump}, G_3, r_3, \mathit{On})\}$, *where*
  $G_1 := h \geq 7$
  $G_2 := h > 4$
  $G_3 := h = 0 \vee x = 3\}$
  $r_1 = (h := h + 0.5,\ x := x)$
  $r_2 = (h := h,\ x := x)$
  $r_3 = (h := h,\ x := 0)$,

Figure 2.4: Water Tank Automaton



- *Act*:
  $Act(On) := \begin{pmatrix} \dot{h} = 1 - 0.0295\sqrt{h} \\ \dot{x} = 0 \end{pmatrix},$
  $Act(Off) := \begin{pmatrix} \dot{h} = -0.0295\sqrt{h} \\ \dot{x} = 0 \end{pmatrix},$
  $Act(Open) := \begin{pmatrix} \dot{h} = -1.5069\sqrt{h} \\ \dot{x} = 1 \end{pmatrix},$

- *Inv*:
  $Inv(On) := h \leq 9.5$
  $Inv(Off) := h \geq 1$
  $Inv(Open) := h \geq 0 \wedge x \leq 3$

Now that we have defined the syntax of a hybrid automaton, we still need to define its semantics.

2.5 Semantics

Figure 2.5 illustrates the content of this section. We are going to see how transition systems can describe the semantics of hybrid automata.
In the following section we define the semantics of a hybrid automaton as a transition system. At first a satisfaction relation for invariants and guards is defined and after that it can be seen how a transition system for a hybrid automaton is built.

2.5.1 Satisfaction Relation for Invariants and Guards

Next we define the satisfaction relation for invariants and guards. We will use the grammar for conditions and terms from Section 2.4, because it defines the form of invariants and guards.
To determine whether the system fulfills a condition, we need to define the valuation of the continuous variables.

Definition 2.2 *Valuations*
A valuation v for the set of continuous variables X is a function $v\colon X \to \mathbb{R}$, assigning to each variable $x \in X$ its current value $v(x)$. We denote the set of all valuations with V.

In the following, t_1, t_2 are terms and *cond*, *cond*$_1$, *cond*$_2$ are conditions.
Now we can define whether a condition *cond* holds for a valuation v, also written as $v \models \mathit{cond}$. We will do this by induction.
First, we need to define the evaluation of terms:

$$[\![c]\!] = c,$$

$$[\![x]\!] = v(x),$$

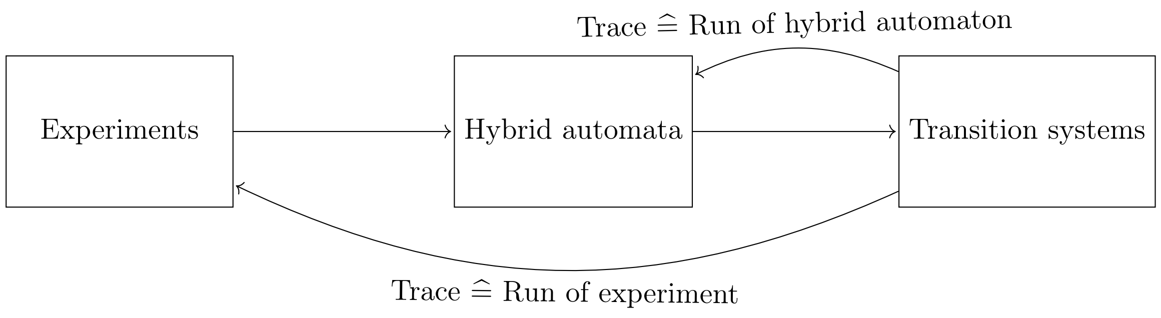
$$[\![t_1\ \gamma\ t_2]\!] = [\![t_1]\!]\ \gamma [\![t_2]\!] \text{ where } \gamma \in \{+, *, -\}.$$

Now we can define the relation $\models$:

$$\begin{array}{ll} v \models t_1\ \alpha\ t_2 & \text{iff } [\![t_1]\!]\ \alpha\ [\![t_2]\!], \text{ where } \alpha \in \{=, <, >, \leq, \geq\}, \\ v \models \mathit{cond}_1 \wedge \mathit{cond}_2 & \text{iff } v \models \mathit{cond}_1 \text{ and } v \models \mathit{cond}_2, \\ v \models \mathit{cond}_1 \vee \mathit{cond}_2 & \text{iff } v \models \mathit{cond}_1 \text{ or } v \models \mathit{cond}_2, \\ v \models \neg \mathit{cond} & \text{iff } v \not\models \mathit{cond}. \end{array}$$

Finally the invariant of a location $l \in L$ holds for a valuation $v \in V$, if $v \models Inv(l)$. The guard G of a transition $e = (l, a, G, r, l') \in E$ holds for a valuation $v \in V$ when $v \models G$.

Figure 2.5: Experiments, Hybrid Automata and Transition Systems



We still need to determine how to evaluate the result of the resets from r. The result is a new valuation v'. We know that r is a list of resets of the form $x := t$, where t is a term. We define a function $\hat{r}$ for a list of resets r that gives the corresponding reset $\hat{r}(x)$ for a variable $x \in X$.
Now we can evaluate a list of resets r, by evaluating the reset $\hat{r}(x)$ for each variable x by

$$[\![\hat{r}(x)]\!] = [\![t]\!].$$

The new valuation v' is determined by

$$v'(x) = [\![\hat{r}(x)]\!] \text{ for each } x \in X.$$

2.5.2 Transition System Semantics of a Hybrid Automaton

Finally we will define the semantic of a hybrid automaton, as a transition system. It needs to be defined how the states of this transition system look like. As mentioned before, the states of a hybrid automaton contain discrete and continuous parts. So the states of the corresponding transition system are defined as a set $S = L \times V$. Such a state $s = (l, v) \in S$ consists of the location $l \in L$ for the discrete part and a valuation $v \in V$ for the continuous part of the state.

Let $H = (L, X, Lab, E, Act, Inv)$ be a hybrid automaton. The corresponding transition system $T_H = (S, S_0, Lab_{T_H}, \rightarrow)$ is defined as

- $S = L \times V$
- $S_0 \subseteq S$, contains all states $s \in S$ in which the hybrid system H can be at the beginning of the execution
- $Lab_{T_H} = Lab \cup \mathbb{R}_{\geq 0}$, Lab for the discrete and $\mathbb{R}_{\geq 0}$ for the timed transitions

- the transition relation $\rightarrow$ is separated in two types of transitions:
  - discrete transitions: $(l, v) \rightarrow^a (l', v')$, for $a \in \mathit{Lab}$, if the following conditions hold:
    - $(l, a, G, r, l') \in E$
    - $v \models G$
    - $v'(x) = [\![r(x)]\!]$, for all $x \in X$
    - $v' \models \mathit{Inv}(l')$
  - timed transitions: $(l, v) \rightarrow^t (l, g(t))$, for $t \in \mathbb{R}_{\geq 0}$, if the following conditions hold:
    - $g = \mathit{Act}(l)(v)$
    - $g(t') \models \mathit{Inv}(l)$, for $0 \leq t' \leq t$

Note, to keep things simple, we use v in two ways. In the states (l, v), v is a function with domain X that is taking values in $\mathbb{R}$. If we want to use it for the activity, actually an element from $\mathbb{R}^{|X|}$ is needed. A function $v \colon X \rightarrow \mathbb{R}$ can also be seen as such a tuple of its values. Therefore we will also use v in this way.

For a discrete step $(l, v) \rightarrow^a (l', v')$ a transition $e = (l, a, G, r, l') \in E$ needs to exist in the underlying hybrid system, the invariant of the location l needs to hold for v on the one hand and the guard G on the other hand to ensure that the transition is enabled, v' is calculated by the reset and needs to satisfy the invariant of the target location l'.
With a timed transition $(l, v) \rightarrow^t (l, f(v, t))$ the system stays in a location l if the function f is the action of the location l and for all $0 \leq t' \leq t$, $f(v, t')$ satisfies the invariant of the location l.
As $t \in \mathbb{R}_{\geq 0}$ is real valued, there are uncountably many edges from the state (l, v) of the form $(l, v) \rightarrow^t$ as long as v satisfies the invariant of l.

For the timed transitions we have to look at the way the continuous part of the target state is calculated. So for all $x \in X$, beginning at time $t_0 = 0$ with the initial state $(l_0, v_0) \in S_0$, where $v_0(x) = x_0$ and $\mathit{Act}(l_0)(v_0) = g$, the value of the continuous variable x flows according to the related first order ODE where we know that g is the solution of the ODE and the initial value is $g(0) = x_0$.
For all other states $(l, v) \notin S_0$ the initial value is $g(0) = v(x)$. How a calculation of the solution and the new value works can be seen in the following example.

Example 2.4 *We can now state the transition system for the water tank example from Figure 2.4.* $T_{H_{\text{WaterTank}}} = (S, S_0, Lab_T, \rightarrow)$, *where*

- $S = \{(On, v) \mid v(h), v(x) \in \mathbb{R}_{\geq 0}\} \cup \{(Off, v) \mid v(h), v(x) \in \mathbb{R}_{\geq 0}\} \cup \{(Open, v) \mid v(h), v(x) \in \mathbb{R}_{\geq 0}\}$
- $S_0 = \{(On, v_0)\}$ *where* $v_0 \colon v_0(x) = 0$ *and* $v_0(h) = 1$
- $Lab_T = \{\textit{Turn pump off}, \textit{Open valve}, \textit{Connect pump}\} \cup \mathbb{R}_{\geq 0}$
- $\rightarrow =$
$\Big\{\Big((On, v), t, (On, f(v,t))\Big) \mid v(h), f(v,t)(h) \leq 9.5 \wedge f = Act(On)\Big\} \cup$
$\Big\{\Big((On, v), \textit{Turn pump off}, (Off, v')\Big) \mid 7 \leq h \leq 9 \wedge v'(h) = v(h) + 0.5\Big\} \cup$
$\Big\{\Big((Off, v), t, (Off, f(v,t))\Big) \mid v(h), f(v,t)(h) \geq 1 \wedge f = Act(Off)\Big\} \cup$
$\Big\{\Big((Off, v), \textit{Open valve}, (Open, v')\Big) \mid 1 \leq h < 4\Big\} \cup$
$\Big\{\Big((Open, v), t, (Open, f(v,t))\Big) \mid v(h), f(v,t)(h) \geq 0 \wedge v(x), f(v,t)(x) \geq 0 \wedge f = Act(Open)\Big\} \cup$
$\Big\{\Big((Open, v), \textit{Connect pump}, (On, v')\Big) \mid v(h), v(x) = 0 \wedge v'(x) = 0\Big\}$

To see how the calculation of the actions works, here is an example for a timed transition from the state (Off, v) *where* $v(h) = 6$.
First we have to find a general solution for

$$\dot{h} = -0.0295\sqrt{h},$$

which is

$$h(t) = \tfrac{1}{4}(-0.0295 \cdot t + C)^2.$$

Including the initial value $h(0) = 6$ *gives*

$$\begin{gathered} 6 = \tfrac{1}{4}(-0.0295 \cdot 0 + C)^2 \\ \sqrt{24} = C, \end{gathered}$$

which can be used to get the particular solution for this initial value,

$$h(t) = \tfrac{1}{4}(-0.0295 \cdot t + \sqrt{24})^2.$$

We now choose the time as $t = 60$, then

$$h(60) = \frac{1}{4}(-0.0295 \cdot 60 + \sqrt{24})^2 \approx 2.4476.$$

So there is a timed transition

$$(\mathit{Off}, v) \rightarrow^{60} (\mathit{Off}, v'), \text{ where } v(h) = 6 \text{ and } v'(h) \approx 2.4476$$

in the transition system of the water tank. And as $1 \leq v(h) < 4$, the transition $e = (\mathit{Off}, \mathit{Open\ valve}, G, r, \mathit{Open})$, with $G = \{v \mid v(h) > 4\}$ and $r\colon r(h) = h, r(x) = x$, in the water tank automaton is enabled. So there is also a discrete transition

$$(\mathit{Off}, v') \rightarrow^{\mathit{Open\ valve}} (\mathit{Open}, v').$$

2.6 Ordinary Differential Equations

As mentioned in Section 2.4 the activity of a location is described by a system of first order ordinary differential equations, because they have a lot of applications in scientific fields. For a better understanding of the evolution of a hybrid system this Section will give a quick overview of ordinary differential equations. The information in this section are taken from [Col81] and [Grö77].

2.6.1 Definition

A differential equation is a mathematical equation that contains derivatives of a function which is the unknown variable in that equation. If this function is a function of only one independent variable the differential equation is called to be **ordinary** and is abbreviated as ODE. A differential equation is, in contrast to an ordinary one, called partial differential equation if the unknown variable is a function of multiple independent variables.
For $n \in \mathbb{N}$, $D \subset \mathbb{R}^{n+2}$ and a function $F\colon D \to \mathbb{R}$ an nth order ODE where the unknown function is $y(x)$, is in the implicit form when it is defined as

$$F(x, y(x), \dot{y}(x), \ddot{y}(x), ..., y^{(n)}(x)) = 0.$$

In this paper the explicit form is used. For $f\colon \mathbb{R}^{n+1} \to \mathbb{R}$, an ODE is in the explicit form, if the equation is resolved to its highest derivative:

$$y^{(n)}(x) = f(x, y(x), \dot{y}(x), \ddot{y}(x), ..., y^{(n-1)}(x))$$

The highest derivative determines the **order** of the differential equation. Thus, a first order ordinary differential equation in explicit form looks like

$$\dot{y}(x) = f(x, y(x)),$$

which can also be written shortly as

$$\dot{y} = f(x, y).$$

The latter form is mostly used throughout the paper.
As previously mentioned we use a **system of first order ODEs** for the activity.

2.6.2 Existence and Uniqueness of solutions

For a system of n first order ODEs $\dot{y_1}, ..., \dot{y_n}$

$$\begin{gathered}\dot{y_1} = f_1(x, y_1, y_2, ..., y_n)\\ \dot{y_2} = f_2(x, y_1, y_2, ..., y_n)\\ \vdots\\ \dot{y_n} = f_n(x, y_1, y_2, ..., y_n)\end{gathered}$$

and a set of initial values

$$y_i(a) = b_i \text{ (for } i = 1...n),$$

where $a, b_i \in \mathbb{R}$, the essential question is, whether there is a solution to this system and if it is the only one. This solution would be a system of functions that satisfy the ODEs and the initial values.
According to [Col81], for a system of n first order ODEs the existence and uniqueness of the solution system is given, if all functions f_i (for $i = 1...n$) are continuous in the range

$$|x - a| < h, |y_i - b_i| < hM_i$$

for some positive numbers h and M_i, and meet the conditions

$$|f_i(x, y_1, y_2, ..., y_n)| \leq M_i$$

(Boundedness) and

$$\begin{gathered}|f_i(x, y_1, y_2, ..., y_n) - f_i(x, y{*}_1, y{*}_2, ..., y{*}_n)| \\ \leq k\{|y_1 - y *_1| + ... + |y_n - y *_n|\}. \\ \text{(Lipschitz condition)}\end{gathered}$$

for some positive constant k and two different points $\{x, y_1, y_2, ..., y_n\}$, $\{x, y{*}_1, y{*}_2, ..., y{*}_n\}$ with the same x-coordinate.

3 The syntax and semantics of SoPHY

In the following chapter we will define the language SoPHY, a specification language for hybrid automata. Section 3.2 is devoted to the syntax. After that in Section 3.3 we will see how a hybrid automaton can be built from a specification in SoPHY. In the whole chapter the water tank example from the previous sections is used a the better understanding of the concept.

3.1 Challenges

The aim of SoPHY is to provide a language to model scientific experiments with hybrid systems in a didactical setting such as biology class in school. Every experiment has an experimental arrangement that describes the scientific system and contains a set of variables that change their values during the experiment. In an application that models scientific experiments, it mainly needs to be possible for the teachers to define those variables and the scientific laws they follow. They also need to be able to determine the experimental arrangement in a way that fits to their purpose. If they want to examine a special detail in the experimental arrangement, they should be able the place this detail in the center of the experiment. According to this point the language should give the teacher a great degree of freedom to design the experiment. In the last chapters, we have seen the important features of experiments and how they can be modeled with hybrid systems. SoPHY needs to be able to represent all these features. Since SoPHY is meant to be used in the didactic environment, the syntax should be easy to read and does not need to correspond exactly to those of hybrid systems.

3.2 Syntax

The format on which the syntax of SoPHY is based is JSON. The JavaScript Object Notation, short JSON, is a text format for the serialization of structured data [Cro06]. This format has been chosen because it is easy to understand for users and computers. As we stick to the conventions of JSON when designing the syntax, it is not necessary to write a parser for SoPHY in future tools. There are parsers for JSON in almost all

popular programming languages, which is also making it easier to use SoPHY on multiple platforms.
In SoPHY the specification of an experiment is separated into two parts. The first is called the *Definition part.* The general information about the structure of the experiment is defined in this part, by indicating the characteristics like properties and events that can happen. In addition to this, the *Configuration part* configures the exact experiment by determining the initial values of the system.

3.2.1 Definition part

To make SoPHY easy to understand, we have aligned the structure of the definition to the characteristics of experiments that we have elaborated in the course of the work so far. To this end, we have defined a number of key words that should guide the user through the structure of the experiment.
At the very beginning, the `name` of the experiment is set. Looking at the previous examples, we first defined the `properties`, which contain values of the experiment, which are continuously changing. After that, we devoted attention to the discrete `states`, which describe the status of the experiment. Each state is identified by a `name`. We have found that the current state of the experiment depends on the values of the properties. Therefore, a `range` can be defined for each state, which limits the values of the properties for this state. The `evolution` of the properties depends on the respective state and can be defined there. In examining the experiments we have observed that the discrete state changes due to different `events`. These events can be described by a `name`. When such an event occurs depends on the values of the properties and is determined by a `condition`. Such an event `results` in another state that has a `name`. In addition, the values of the properties, `property_values` can be changed during an event.
An example for a definition part in SoPHY can be found in Listing 3.1. In this example the water tank from Figure 2.4 is defined. The `properties` of the water tank are the height of the water level h and the clock x. The `states` show in which different states the pump can be. The pump can either be *On*, *Off* or the valve can be *Open.* We will now look at the components of the state *On* to explain the syntax. The same scheme can also be applied to the other states. The pump is *On* as long as the height does not reach 9.5cm. The height evolves due to the differential equation $h' = 1 - 0.0295 * sqrt(h)$ (sqrt denotes the square root) and the clock due to $x' = 0$.
In the *On* state only one `event` can happen. The pump can be turned off. The `condition` for this event is, that the water level raised up to 7cm. The `result` of this event is the state *Off*, while the water level h is increased by 0.5cm and the clock x does not change.

Listing 3.1: Definition part example

```
{
    "name":"Water tank",
    "properties":[
        "h",
        "x"
    ],
    "states":[
        {
            "name":"On",
            "range":"h <= 9.5",
            "evolution":"h' = 1 - 0.0295 * sqrt(h); x' = 0",
            "events":[
                {
                    "name":"Turn pump off",
                    "condition":"h >= 7",
                    "result":{
                        "name":"Off",
                        "property_values":"h := h + 0.5; x := x"
                    }
                }
            ]
        },
        {
            "name":"Off",
            "range":"h >= 1",
            "evolution":"h' = -0.0295 * sqrt(h); x' = 0",
            "events":[
                {
                    "name":"Open valve",
                    "condition":"h < 4",
                    "result":{
                        "name":"Open",
                        "property_values":"h := h; x :=x"
                    }
                }
            ]
        },
        {
            "name":"Open",
            "range":"h >= 0 && x <= 3",
            "evolution":"h' = -1.5069 * sqrt(h); x' = 1",
            "events":[
                {
                    "name":"Connect pump",
                    "condition":"h = 0 || x = 0",
                    "result":{
                        "name":"On",
                        "property_values":"h := h; x := 0"
                    }
                }
            ]
        }
    ]
}
```

To specify the syntax formally, we have generated a grammar on the basis of which a definition of an experiment in SoPHY is constructed. This is the grammar of the definition part, written in EBNF:

⟨*System*⟩ = '{'
    "name": ⟨*Identifier*⟩ ,
    "properties": ⟨*PropertyList*⟩ ,
    "states": '[' (⟨*State*⟩)+ ']'
    '}'

⟨*State*⟩ = '{'
    "name": ⟨*Identifier*⟩ ,
    "range": ⟨*Condition*⟩ ,
    "evolution": ⟨*Activity*⟩ ,
    "events": '[' (⟨*Event*⟩)+ ']'
    '}'

⟨*Event*⟩ = '{'
    "name": ⟨*Identifier*⟩ ,
    "condition": ⟨*Condition*⟩ ,
    "result": ⟨*Result*⟩
    '}'

⟨*Result*⟩ = '{'
    "name": ⟨*Identifier*⟩ ,
    "property_values": ⟨*Resets*⟩
    '}'

⟨*Identifier*⟩ = ⟨*Letter*⟩ (⟨*Letter*⟩ | ⟨*Digit*⟩ | '_')*

⟨*PropertyList*⟩ = '[' (⟨*Identifier*⟩ ,)* ⟨*Identifier*⟩ ']'

⟨*Condition*⟩ = ⟨*Expression*⟩ (= | < | > | <= | >=) ⟨*Expression*⟩ | ⟨*Condition*⟩ AND ⟨*Condition*⟩ | ⟨*Condition*⟩ OR ⟨*Condition*⟩ | NOT ⟨*Condition*⟩

⟨*Expression*⟩ = arithmetical expressions with +, -, *, real constants and the properties from ⟨*PropertyList*⟩

⟨*Activity*⟩ = (⟨*ODE*⟩ ;)* ⟨*ODE*⟩

⟨*ODE*⟩ = first order ordinary differential equation in explicit form

⟨*Resets*⟩ = (⟨*Reset*⟩)*

⟨*Reset*⟩ = ⟨*Identifier*⟩ ':' '=' ⟨*Expression*⟩

⟨*Letter*⟩ = 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G'
| 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N'
| 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U'
| 'V' | 'W' | 'X' | 'Y' | 'Z' | 'a' | 'b'
| 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i'
| 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p'
| 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w'
| 'x' | 'y' | 'z'

⟨*Digit*⟩ = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

We now want to explain this grammar in a textual way.
The system is identified by a name and has a nonempty list of properties that define the behavior of the system. Such a system consists of a nonempty set of states in which the system can be. The states are identified by a name and determine a range in which the properties can be, without changing the state. The evolution in the state contains a nonempty list of first order ordinary differential equations that describe the evolution of the properties.
Each state has a set of events that show what can happen in the related state. An event has a name and a condition that determines when this event happens. The result of the event consists of the resulting state and the new values for the properties.

3.2.2 Configuration part

Also in the configuration of the experiment, keywords, which are based on our gained knowledge about experiments, are used. First of all, the name used in the definition is needed as the configurations `id`, to connect the two parts. In order to distinguish different configurations of the same experiment, a configuration can be provided with a `name`. To describe the starting point of an experiment, the `initial_state`, in which the system starts, must be specified. In addition, `initial_values` for all properties must be defined, whereby the initial description of the experiment is complete.
An example for a configuration part in SoPHY can be found in Listing 3.2. In this example

the created water tank starts in the state *On*, so the pump is working. The initial water level is 1cm and the clock is set to 0.

Listing 3.2: Configuration part example

```
{
    "id":"Water Tank",
    "name":"Water Tank",
    "initial_state":"On",
    "initial_values":[
        1,
        0
    ]
}
```

For the configuration, we have again specified the syntax formally by generating a grammar, on whose basis a configuration in SoPHY is constructed. This is the grammar of the configuration part, written in EBNF.

⟨*Setup*⟩ = '{'
    "id": ⟨*Identifier*⟩ ,
    "name": ⟨*Identifier*⟩ ,
    "initial_state": ⟨*Identifier*⟩ ,
    "initial_values": ⟨*ValueList*⟩ '}'

⟨*Identifier*⟩ = ⟨*Letter*⟩ (⟨*Letter*⟩ | ⟨*Digit*⟩ | '_')*

⟨*ValueList*⟩ = '[' (⟨*Value*⟩ ,)* ⟨*Value*⟩ ']'

⟨*Value*⟩ = ⟨*DigitSequence*⟩ ('.' ⟨*DigitSequence*⟩)?

⟨*Letter*⟩ = 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G'
    | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N'
    | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U'
    | 'V' | 'W' | 'X' | 'Y' | 'Z' | 'a' | 'b'
    | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i'
    | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p'
    | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w'
    | 'x' | 'y' | 'z'

⟨*Digit*⟩ = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

⟨*DigitSequence*⟩ = ⟨*Digit*⟩ (⟨*Digit*⟩)*

In summary, the grammar can be explained as follows.
The setup is referred to a system from the definition part by an ID that needs to match the name of the system. The system has an own name to specialize it. The initial state refers to one of the states from the system in the definition part and determines that this system starts in this actual state. A setup also has a nonempty list of initial values that assigns a value to each property of the system.

3.3 Semantics: From SoPHY to hybrid automata

A hybrid automaton can be extracted from each pair of a definition and a configuration part, a. In the following section we will show how. We will use the definition part of SoPHY to create an automaton by using the definition of a hybrid automaton from Section 2.4 and we will add the initial values, which are important for the simulation, with the configuration part.
In this section we will see expressions such as: $x\ in$ `"properties"`. This means that x is an element associated with the key `"properties"` in SoPHY. This expression will be used to build a set containing all these x. In addition, we will also use expressions of the form: `"states"."name"`. These result from the hierarchically nested structure of SoPHY and are used to access the names of all states.

3.3.1 Continuous Variables

The continuous variables are the global `properties` of a hybrid automaton. Therefore they are given at the very beginning of the definition part, right after the name of the system. So the finite set X is defined by:

$$X = \{x \mid x \in \texttt{"properties"}\}$$

In the example in Listing 3.1 a water tank has the properties h and x.

3.3.2 Locations

In the definition part each system has a set called `states` which describe the various states in which the system can be. These states build the locations of the hybrid automaton. For each state a location is created:

$$L = \{l \mid l \in \texttt{"states"."name"}\}$$

Figure 3.1: Locations

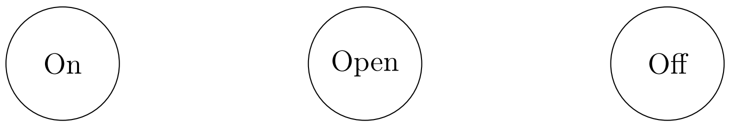
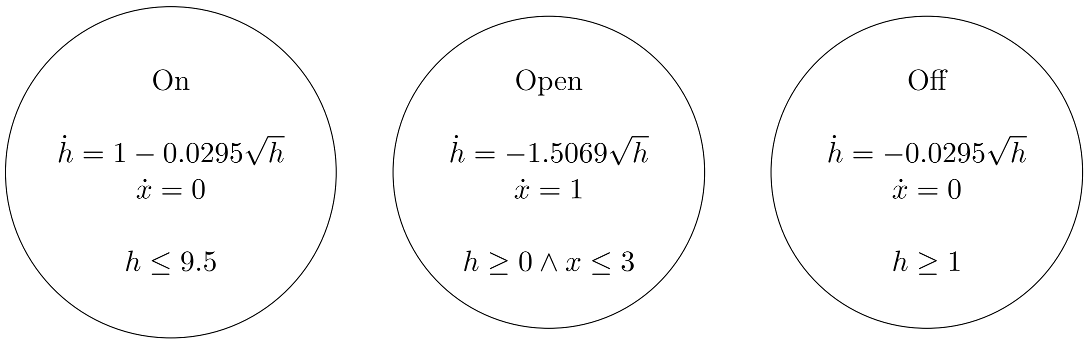


Figure 3.2: Invariants and Continuous Evolution


On
$\dot{h} = 1 - 0.0295\sqrt{h}$
$\dot{x} = 0$
$h \leq 9.5$
Open
$\dot{h} = -1.5069\sqrt{h}$
$\dot{x} = 1$
$h \geq 0 \wedge x \leq 3$
Off
$\dot{h} = -0.0295\sqrt{h}$
$\dot{x} = 0$
$h \geq 1$

We can now begin to create the graphical representation of the hybrid automaton. The locations of the water tank example can be seen in Figure 3.1.

3.3.3 Invariants and Continuous Evolution

The invariant of an location describes for which condition the system can stay in this location. In the definition part each state has a `range`. This is used to assign an invariant to each location. The function *Inv* is determined by:

$$Inv(s.\texttt{"name"}) \mapsto (s.\texttt{"range"}), \text{ where } s \in \texttt{"states"}.$$

The activity of a location describes the evolution of the properties in the respective location. This `evolution` is also assigned to a state in the definition part. The function *Act* is determined by the definition part in the following way:

$$Act(s.\texttt{"name"}) \mapsto (s.\texttt{"evolution"}), \text{ where } s \in \texttt{"states"}.$$

The current progress of the automaton is shown in Figure 3.2.

3.3.4 Transitions

The next step is to connect the locations with transitions. This information is called `events` in the definition part. Each state has a set of events that can happen. Such an event describes a discrete change that can happen to the system. An event has a `name` that is going to be the label of the transition. The set of labels is filled by

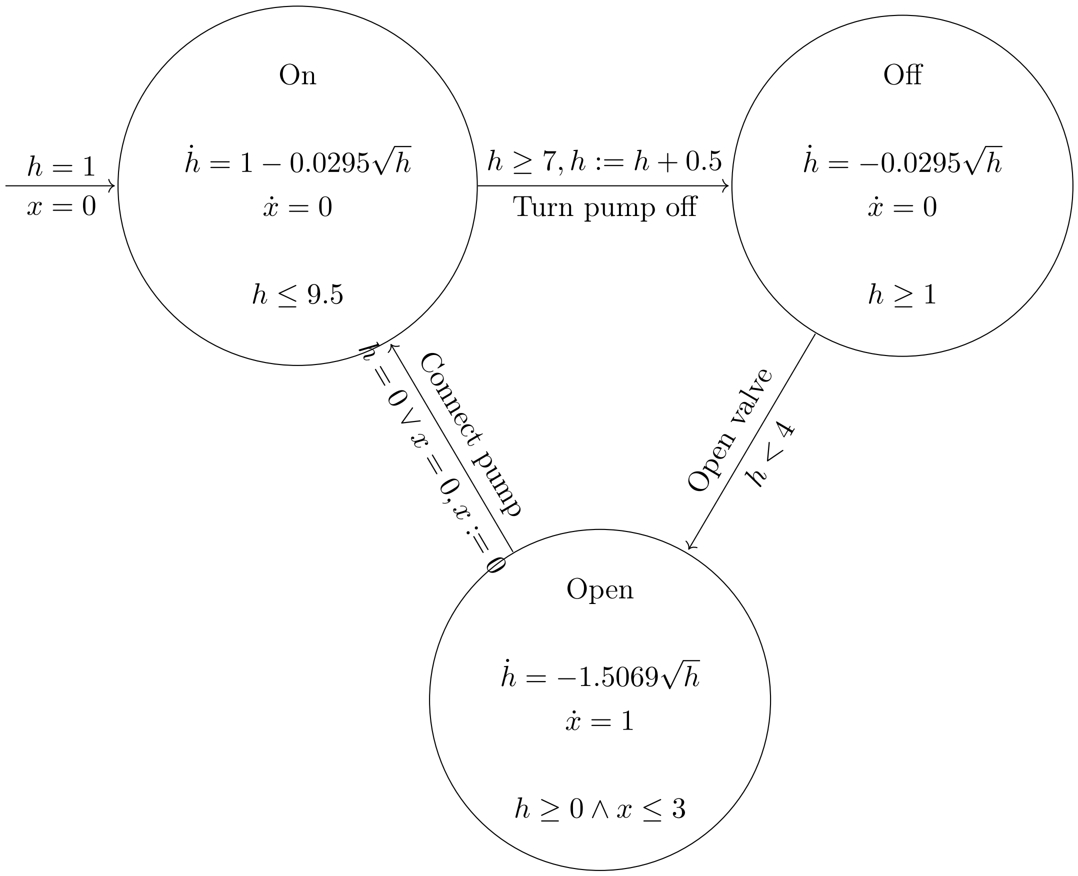
$$Lab = \{\texttt{"states"}.\texttt{"events"}.\texttt{"name"}\}.$$

Furthermore the events have a `condition` which determines when this actual event happens. This is the guard of the transition. Each event results in another state of the system. This `result` contains the `name` of the next state and the new values of the properties, called `property_values`, which denote the resets of the transition. The name of the next state is the target location, while the state that contains the event is the source location. Therefore the set of transitions is

$$\begin{aligned}&E = \{s.\texttt{"name"},\\ &s.e.\texttt{"name"},\\ &s.e.\texttt{"condition"}\\ &s.e.\texttt{"result"}.\texttt{"property\_values"},\\ &s.e.\texttt{"result"}.\texttt{"name"}\},\\ &\text{where } s \in \texttt{"states"} \text{ and } e \in \texttt{"events"}.\end{aligned}$$

In Figure 3.3 can be seen how the locations are connected to each other and how this transitions are annotated with labels, guards and resets.

Figure 3.3: Transitions and Initial State



3.3.5 Initial States

Now the hybrid automaton from the definition part is completed. We still have to include the information from the configuration part, which determines how the actual system is configured. The configuration part contains the setup which finalizes the system. The automaton will be extended by an edge that defines an `initial_state` and the `initial_values` of the properties. The final system can be seen in Figure 3.3.

4 Using SoPHY: Scientific examples

In this Section we will see how to model experiments with SoPHY.
At first a physical example is given. In school pendulum experiments are used to learn about the effects of velocity and gravity. We will see how to specify a pendulum with SoPHY, which is constrained by a pen.
The second example comes from the field of biology. Ecology, a part of biology, is often concerned with questions about population models. We will use SoPHY to specify a predator prey model that can be used to monitor the growth of two competing populations.

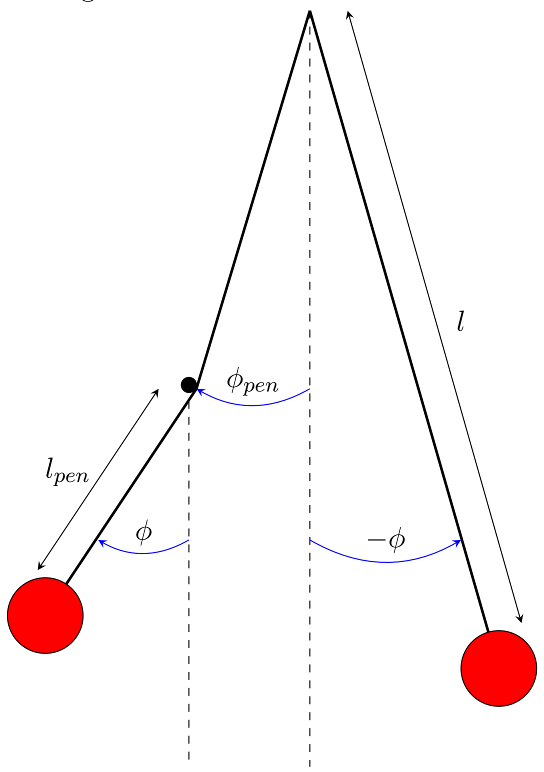
4.1 Constrained Pendulum [Kle08]

The simple pendulum is an experiment that nearly every student has seen in school once. This experiment is used so often, because the mathematical representation, which assumes that there is no friction at all and the mass of the pendulum is concentrated to one point, does not differ from reality so much. To approach this idealized version in a real experiment the students just need to use a long and thin thread to minimize the friction and to use a small and heavy pendulum to approach that the mass is concentrated to one point. In that way the students can calculate the values of the experiment in an easy way without a big loss of accuracy.
Some questions on this experiment are for example

- Does the duration of the oscillation depend on the mass of the pendulum?
- What is the difference in starting at a high and at a low angle?
- What happens if the length of the pendulum is changed?

A modification to the simple pendulum is the **constrained pendulum**. In contrast to the simple one, the constrained pendulum is extended by a pen which the thread hits at some point. A diagram of this pendulum is shown in Figure 4.1. The pendulum has length l and when it hits the pen the length is shortened to l_{pen}. ϕ_{pen} determines the angle of the position of the pen and ϕ is the angle of the pendulum.
To define this experiment with SoPHY, first of all we have to think about the properties of

Figure 4.1: Constrained Pendulum



the systems. The question is how the behavior of the system can be described by variables that change continuous in time. With this pendulum it is the angle ϕ that changes while the pendulum is oscillating and the angular velocity v of the end of the pendulum. So we call the properties of the system ϕ and v.

The next step is to determine the states of the experiment. These states describe the discrete changes which can happen during the experiment. For this experiment the pendulum can be in two states: to be *unconstrained* or *constrained.*

The pendulum is unconstrained if it oscillates without touching the pen. So the range of the state *unconstrained* is $\phi \leq \phi_{pen}$ because as long as the pendulum does not hit the pen, its angle is smaller than the one where the pen is located. While the pendulum is unconstrained the velocity v evolves according to the differential equation $\dot{v} = -g\ sin\ \phi$ and the equation for the evolution of the angle is $\dot{\phi} = \frac{1}{l}v$.

The pendulum is constrained when the pendulum hits the pen, which leads to the range $\phi \geq \phi_{pen}$ for the state *constrained.* As long as the pendulum is constrained by the pen the angle ϕ evolves with $\dot{\phi} = \frac{1}{l_{pen}}v$. This is the same equation as in the *unconstrained* state, but the length of the pendulum is shortened by the pen to l_{pen}. The velocity v evolves in the same way as in the *unconstrained* state, because it does not depend on the length of the pendulum.

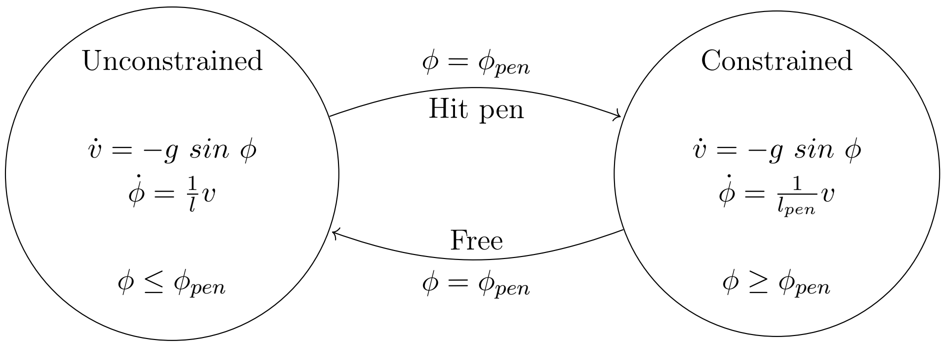
After the states with their range and the evolution of the properties are defined we need to think about the events that can happen in the two states. An event is a discrete change that happens and has the result, that the system changes its state. Regarding the *unconstrained* state, the event that can happen is that the pendulum hits the pen. The

condition of the event is $\phi = \phi_{pen}$ and it results in the *constrained* state, without changing the values of the properties. While the pendulum touches the pen, the state changes to *constrained*. In the moment the pendulum moves away from the pen, when $\phi = \phi_{pen}$, an event occurs that brings the system back to the *unconstrained* state.
We now got all the information to define the experiment with SoPHY. In Listing 4.1 the definition of the pendulum in SoPHY is given. It should be noted, that ϕ_{pen}, g, l and l_{pen} need to be replaced by exact values to describe a particular system. This is not done here, for the reason that the definition does not depend on the value of these variables. The resulting hybrid automaton is shown in Figure 4.2.

Listing 4.1: Pendulum Definition

```
[
   {
      "name":"Constrained Pendulum",
      "properties":[
         "v",
         "phi"
      ],
      "states":[
         {
            "name":"Unconstrained",
            "range":"phi <= phi_pen",
            "evolution":"v'=-g sin phi; phi'=1/l v",
            "events":[
               {
                  "name":"Hit pen",
                  "condition":"phi = phi_pen",
                  "result":{
                     "name":"Constrained",
                     "property_values":"v:=v; phi:=phi"
                  }
               }
            ]
         },
         {
            "name":"Constrained",
            "range":"phi >= phi_pen",
            "evolution":"v'=-g sin phi; phi'=1/l_pen v",
            "events":[
               {
                  "name":"Free",
                  "condition":"phi = phi_pen",
                    "result":{
                        "name":"Unconstrained",
                        "property_values":"v:=v; phi:=phi"
                    }
                }
            ]
        }
    ]
  }
]
```

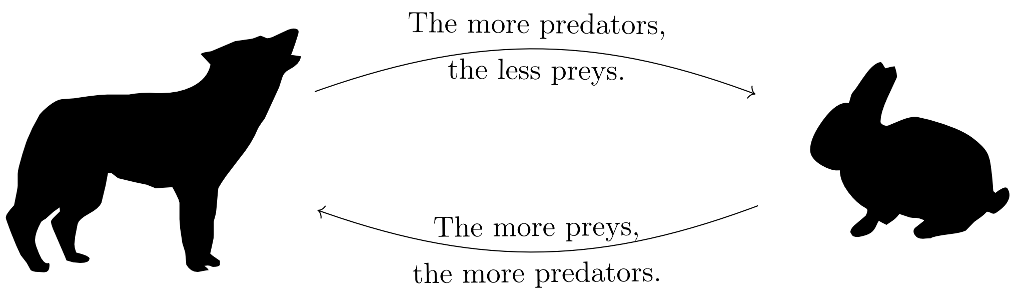
Figure 4.2: Pendulum Automaton



4.2 Predator-Prey Relationship

Here we consider the well known predator-prey relationship [Grü08]. The predator-prey relationship is a model of a section of the food chain, which involves two different types. One species is the predator whose source of food is the other species, the prey. In our model, we assume that the food resources for the prey are unlimited. Figure 4.3 illustrates this relationship. The predator must eat prey in order not to die. The more prey there is,

Figure 4.3: Predator-Prey Model



the faster the predator can multiply. This, however, leads to a reduction in the number of prey, as many predators eat a lot of prey. However, the decreasing number of prey leads to a reduction in the predator population. As a result, the prey population can grow again.
In order to model the predator-prey relationship in SoPHY, we have to think again about the obligatory properties. First of all, the experiment deals with the size of the two populations. We denote the prey population with x_1 and the predator population with x_2. The equations used in the following are from [Grü08]. Both populations grow according to the ODE $\dot{x} = \lambda x$, where λ is the growth rate resulting from the birth rate γ, minus the death rate σ: $\lambda = \gamma - \sigma$.
Let us first consider the prey population. The birth rate γ of the prey population is constant. The death rate is given by $\sigma = \tilde{\sigma} + bx_2$ where $\tilde{\sigma} \in (0, \gamma)$ is the natural death and the term bx_2 describes the death by predator. The constant b depends on the species. We define $a = \gamma - \tilde{\sigma} > 0$, which can also be called the net growth rate of the prey. From this we can derive the following equation:

$$\begin{aligned}
\dot{x_1} &= \lambda x_1 = (\gamma - \sigma)x_1 \\
\dot{x_1} &= (\gamma - (\tilde{\sigma} + bx_2))x_1 \quad && (\sigma = \tilde{\sigma} + bx_2) \\
\dot{x_1} &= (a - bx_2)x_1 && (a = \gamma - \tilde{\sigma}).
\end{aligned}$$

We now consider the predator population. There, the death rate σ is constant and the birth rate γ arises from $\gamma = \tilde{\gamma} + dx_1$. This means that more predators are born when more prey is available. The constant d depends on the species. We define $c = \sigma - \tilde{\gamma} > 0$ and conclude the following equation:

$$\begin{aligned}
\dot{x_2} &= \lambda x_2 = (\gamma - \sigma)x_2 \\
\dot{x_2} &= (\tilde{\gamma} + dx_1 - \sigma)x_2 \quad && (\gamma = \tilde{\gamma} + dx_1) \\
\dot{x_2} &= (-c + dx_1)x_2 && (c = \sigma - \tilde{\gamma}).
\end{aligned}$$

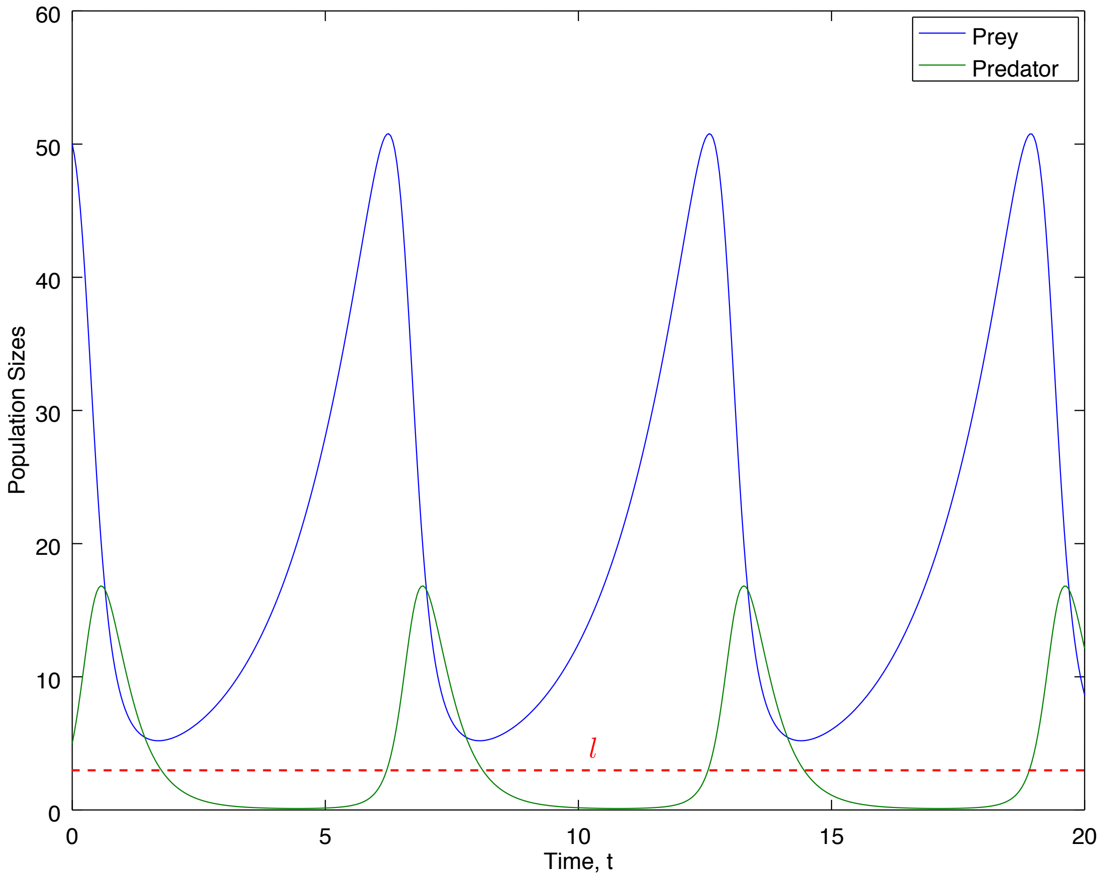
Thus the system of differential equations, which is called *Lotka-Volterra model*, is obtained:

$$\dot{x_1} = ax_1 - bx_1x_2$$

$$\dot{x_2} = -cx_2 + dx_1x_2.$$

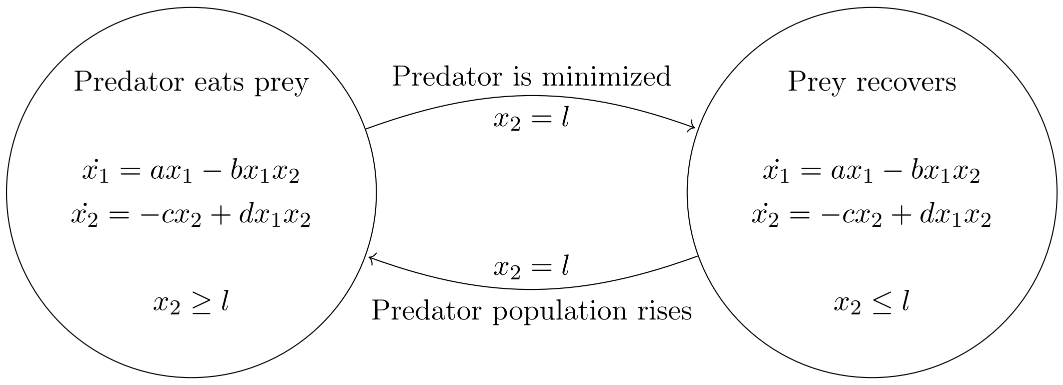
We now need to define the discrete states. In this experiment, the development of continuous variables is not affected by discrete events. This does not mean that no discrete states can or should be modeled. As already mentioned, recurring curves, in which the predator population decreases, occur in the predator-prey model. An example of the Lotka-Volterra model is shown in Figure 4.4. There is a limit l, where the prey population begins to recover as soon as the predator population drops below it. If the predator population exceeds the line again, it has recovered so far that the prey population begins to decline again. This situation can be used as discrete states with the help of this limit. We thus

Figure 4.4: Lotka-Volterra Graph



get two states: *Predator eats prey* and *Prey recovers*. As soon as the predator population reaches the limit in the *Predator eats prey* state, the state changes to *Prey recovers* and vice versa. This also leads to the ranges of the states.
Now we can specify the system in SoPHY. The definition can be found in Listing 4.2. To create a particular experiment, the variables a, b, c, d and l need to be filled with real values. These values depend on the species used in the experiment. The associated hybrid automaton can be found in Figure 4.5.

Figure 4.5: Predator-Prey Automaton



Predator eats prey
$\dot{x_1} = ax_1 - bx_1x_2$
$\dot{x_2} = -cx_2 + dx_1x_2$
$x_2 \geq l$
Predator is minimized
$x_2 = l$
$x_2 = l$
Predator population rises
Prey recovers
$\dot{x_1} = ax_1 - bx_1x_2$
$\dot{x_2} = -cx_2 + dx_1x_2$
$x_2 \leq l$

Listing 4.2: Predator-Prey Definition

```
{
    "name":"Predator Prey",
    "properties":[
        "x1",
        "x2"
    ],
    "states":[
        {
            "name":"Predator eats Prey",
            "range":"x2 >= 1",
            "evolution":"x1=a*x1-b*x1*x2; x2=-c*x2+d*x1*x2",
            "events":[
                {
                    "name":"Predator is minimized",
                    "condition":"x2 = 1",
                    "result":{
                        "name":"Prey is recovering",
                        "property_values":"x1 = x1; x2 = x2"
                    }
                }
            ]
        },
        {
            "name":"Prey is recovering",
            "range":"x2 <= 1",
            "evolution":"x1=a*x1-b*x1*x2; x2=-c*x2+d*x1*x2",
            "events":[
                {
                    "name":"Predator population rises",
                    "condition":"x2 = 1",
                    "result":{
                        "name":"Predator eats Prey",
                        "property_values":"x1 = x1; x2 = x2"
                    }
                }
            ]
        }
    ]
}
```

5 From SoPHY to HSolver

HSolver is a tool for the verification of hybrid systems which can handle, unlike many other tools, non-linear ordinary differential equations. As explained, SoPHY is a specification language for specify hybrid systems with the goal of simulation. Another interesting subject despite the simulation of hybrid systems is safety verification and that is what HSolver does. To now be able to also verify the systems that are specified with SoPHY this chapter will show how to transfer SoPHY code to HSolver.

5.1 Reasons to transfer

In the simulation of experiments, there is usually not only a single way to perform the experiment, but different decisions can be made during the simulation which lead to different further courses of the experiment. In the water tank experiment, for example, the pump can be switched off at $h = 7$ or only at $h = 9.5$. This decision leads, as in the transition system in Figure 2.2, to different further courses of the experiment.
When simulating, students can test whether their assumptions are valid or not, by trying out different runs. However, they can only guess that because they only know that their assumption applies in exactly the runs of the system they have also tried. With the help of verification, it can really be proved if an assumption is valid. Which assumptions could be made in the case of the water tank experiment and how the verification works for such assumptions is explained below.

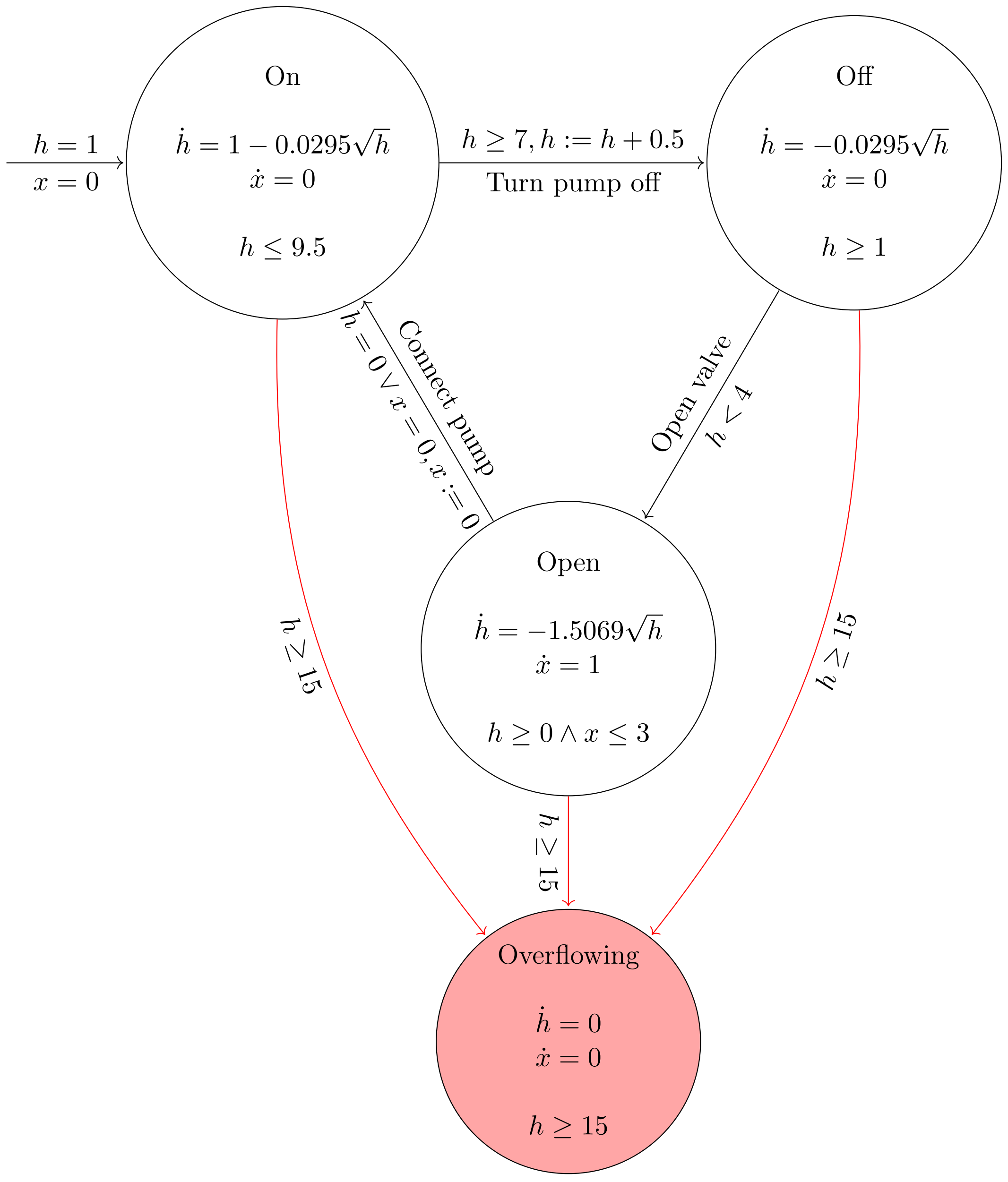
5.1.1 Verification

The matter of safety verification is to check if a system fulfills a given specification. Therefore, the whole system needs to be examined with respect to a global safety property. A safety property asserts, that nothing bad will happen during the evolution of a system [HHWt97]. By now verification is an important part in the development of systems. To determine whether a system is correct or not, a set of properties is created. This properties describe what a system should do and what it should not. If the system fulfills every property, it is called correct. Therefore correctness is not a global property of a system but depends on a specification that is described by multiple properties [BK08].

5.1.2 Safety Verification of Hybrid Systems

Relating to systems that are modeled by automata—like hybrid systems—safety verification is realized by calculating the reachability of certain set of states inside the transition system of the hybrid system. In simple terms, a set of unsafe states is defined or calculated. Let us call this set U. In addition, all reachable states of the transition system are required, we call this set R. If the set of reachable states R does not contain unsafe states from U, the system is safe. As simple as it sounds, a big problem is to determine the reachable states R. In fact, we even know that the reachability problem, needed to verify safety properties, is in general cases undecidable [HKPV98]. There are certain special cases where it is decidable, but the model of the hybrid systems is severely restricted there. Regardless of this fact, HSolver includes an algorithm that terminates when a given system is secure. For practically relevant problems, the algorithm efficiently terminates [RS07]. In simplest terms, their algorithm builds an abstraction of the actual system so that the following holds: If the abstraction is safe, then the original system is safe. They abstract the states of the system so that they no longer consist of a location and exact values of the variables, but from a location and a hyperrectangle, called box, which represents subset of the original state space of the variables. How the algorithm works in detail, can be read in the same article.
As described in [HHWt97], one way to verification is, to expand the hybrid automatons locations L by control locations L_C. The locations of the new, expanded automaton are $L \cup L_C$. This control locations are build in such way, that they are only reached if something bad or forbidden happens.
For example in the automaton of the water tank a control location could be that the tank is oveflowing. A safety property would be something like $h < 15$. The created location, we call it $Overflowing$, would have the invariant $h \geq 15$, which determines that the new forbidden location is only reached if the safety property is not fulfilled. The extended automaton is shown in 5.1.
If now the reachable states $R \subseteq (L \cup L_C) \times V$ are calculated it can be checked whether the safety property is fulfilled or not. It is fulfilled, if the reachable states do not include any states that contain control locations $l_c \in L_C$. In other words, non of the forbidden control locations should be reachable: $R \cap \{(l, v) \mid l \in L_C\} = \emptyset$. Sometimes the automaton even has to be extended by control variables to check for some safety properties. In many times this control variables are clocks, because it should often be checked if a system lasts too long in some locations. In the water tank example a clock is already used to determine that the pump needs 3s to connect. If this should be a safety property, a new control location could be included that is entered when x is greater than 3.
Another way is to define states that are forbidden in the system, without adding new locations. The way to do this, is to add a so called *state assertion* φ to the automaton. The state assertion φ denotes a function, that assigns a safety condition (in the same form as the invariant of a location) $\varphi(l)$ to each location. This safety condition describes the

Figure 5.1: Water tank with forbidden location



unsafe ranges for the variables. For example we could add the safety condition $h \geq 15$ to every location of the water tank and define φ like

$$\varphi(l) = h \geq 15, \text{ for } l \in \{\text{On}, \text{Off}, \text{Open}\}.$$

This has the same effect as adding the control state mentioned before. The state assertion φ is false for a state (l, a) if the safety condition $\varphi(l)$ is false, when every occurrence of the variables $x_i \in X$ in $\varphi(l)$ is replaced by the values a_i of the state. It works the same way for a true state assertion. If the unsafe states are defined like this, the system can be called safe if for every reachable state of the automatons transition system, the state assertion is false. This can again be realized by the intersection of the reachable states R and the unsafe states $S_{\text{Unsafe}} = \{(l, \varphi(l) \mid l \in L)\}$, that needs to be empty $R \cap S_{\text{Unsafe}} = \emptyset$. Both versions can be realized with HSolver.

5.2 Translating SoPHY to HSolver

Here we focus on the conversion from SoPHY to HSolver. Before an algorithm can be developed, it needs to be checked if it is at all possible to transform hybrid automata from one system to the other. To figure this out we will compare the hybrid system models that SoPHY and HSolver use and figure out how we can transform SoPHY into HSolver. First we look at the syntax of HSolver. For that we consider the water tank example in Listing 5.1, which was specified in HSolver.
The continuous variables are listed in brackets after the keyword `VARIABLES`. It is the same with the locations for the keyword `MODES`. After the keyword `STATESPACE`, a lower and upper limit is specified for each mode, for each variable in brackets. The keyword `INITIAL` forwards a collection of constraints in which an initial condition can be specified for each mode in braces. The structure of the constraints is later defined in Definition 5.1. After the keyword `FLOW`, a list of constraints is assigned to each mode. These can contain the variables and the i-th constraint can contain the ODE for the i-th variable. In this case, `x_d` marks the derivation. The transitions are inserted after the keyword `JUMP`. There, source and target location, separated by an arrow, are specified first and then constraints. The keyword `UNSAFE` is used for the verification and therefore remains empty during the translation. More detailed information can be found in [RDS] and [Rat].

Listing 5.1: Water tank in HSolver

```
VARIABLES [ h, x ]
MODES [ On, Off, Open ]
STATESPACE
  On[[1, 9.5], [0, 0]]
  Off[[1, 7.5], [0, 0]]
  On[[0, 4], [0, 3]]
INITIAL
  On{h=1/\x=0}
FLOW
  On{h_d=1-0.0295 * h^0.5 /\ h <= 9.5}{x_d=0 /\ h <= 9.5}
  Off{h_d=-0.0295 * h^0.5 /\ h >= 1}{x_d=0 /\ h >= 1}
  Open{h_d=-1.5069 * h^0.5 /\ h >= 0 /\ x <= 3}
      {x_d=1 /\ h >= 0 /\ x <= 3}
JUMP
  On->Off{h>=7/\h'=h+0.5}
  Off->Open{h<4}
  Open->On{h=0\/x=0/\x'=0}
UNSAFE
```

5.2.1 Comparing the hybrid systems models

We have seen SoPHYs hybrid automata model in Section 2.4 and in the previous Section the model of HSolver was outlined. This section will be dedicated to the comparison of the two models. On that account we will consider six elements of a hybrid automata model that we have in SoPHY:

- Continuous Variables,
- Discrete Locations,
- Initial state,
- Activities,
- Invariants and
- Transitions

For each element, we examine how we can transfer it to the HSolver model. Below we will often use the constraint, that is defined in HSolver in the following way:

Definition 5.1 *Constraint [Rat]*
In HSolver a constraint can contain

- *the quantifiers* $\forall$ *and* $\exists$*, the operators* $\wedge$*,* $\vee$ *and* $\implies$ *,*
- *the predicates* $<, \leq, >, \geq$ *and* $=$*,*
- *the functions* $+, -, *, \hat{}, sin, cos, exp, asin, acos, atan$ *and* log*,*
- *the variables and*
- *real-valued constants.*

HSolvers constraints are more powerful than the conditions we defined for SoPHY. Everything that can be modeled by a condition in SoPHY, can also be modeled by a HSolver constraint. Since we want to transfer SoPHY to HSolver and not the other way around, there is no issue.

Continuous Variables: The continuous variables in SoPHY exactly correspond to the `VARIABLES` in HSolver.

Discrete Locations: Here it is the same as for the variables: The discrete locations exactly correspond to the `MODES` in HSolver.

Initial state: In SoPHY we only have one initial state, HSolver can have an initial constraint for each mode. So we only need to use just one mode and add the values of SoPHYs initial state.

Activities: The ODEs that describe the activities in SoPHY can be described in HSolver within `FLOW`. The ODEs for the evolution of the individual variables are formulated in the list of constraints at the respective position.

Invariants: At first glance, we do not see a space to formulate invariants in HSolver. But, in HSolvers `FLOW` it is not only possible to formulate the activities. Each element is a constraint, that can also contain the first derivative of the corresponding variable. Thus, the invariant that is defined in SoPHY can simply be added to every flow constraint by connecting it with $\wedge$.

Transitions: The transitions can be easily transferred to HSolvers `JUMP`, because they share the same information, only in another form.

In HSolver, there is another set of information that is needed. It needs a state space for every variable for each mode. This means that HSolver needs to know an upper and a lower bound of the variables for every mode. Unfortunately SoPHY does not provide this kind of information. But in some cases these bounds can be found out by a combination of the initial values, the invariants, the guards and the resets. The method how this can be done will be shown in the Algorithm in the next Section. In the cases where this method is not working, the user could be asked to give some more information.
The next section provides pseudocode for an algorithm that transfers SoPHY to HSolver, by solving the differences like we have stated here.

5.2.2 Pseudocode

The following algorithm translates SoPHY code into code that can be used by HSolver to verify the specified hybrid system. The input to this algorithm are the definition of the SoPHY definition part and the configuration of the SoPHY configuration part.

Procedure 1 Translate SoPHY to HSolver

```
Input: definition, configuration
Output: HSolver code
  variables ← [ ]
  modes ← [ ]
  statespace ← [ ]
  initials ← [ ]
  flows ← [ ]
  jumps ← [ ]
  $d$ ⇐ definition
  for Every element $x$ in $d$.properties do
    variables ← variables.add("$x$")
  end for
  for Every element $s$ in $d$.states do
    modes ← modes.add("$s$.name")
    invariant ← $s$.range
    flows ← flows.add("$s$.name" + "{invariant)
    for Every element $e$ in $s$.evolution do
      e_d ← $e$ where every x' is replaced by x_d, where x is a variable
      flows ← flows.add(/\$e\_d$}")
    end for
    flows ← flows.add(})
    for Every element $e$ in $s$.events do
      r ← $e$.result.property_values where every x: is replaced by x', where x is a variable

      jumps ← jumps.add("$s$.name" + "->" + "$e$.result.name" + "{$e$.condition /\[$r$]}")
    end for
  end for
  for Every element $s$ in $d$.states do
    statespace ← statespace.add($s$[)
    for Every element $x$ in variables do
      min ← search in $s$."range" and in jumps that lead to $s$ for the minimal value of $x$
      max ← search in $s$."range" and in jumps that lead to $s$ for the maximal value of $x$
      statespace ← statespace.add([min,max])
    end for
    statespace ← statespace.add(])
  end for
  initial_values ← [ ]
  for Every element $v$ in $c$.initial_values do
    initial_values ← initial_values.add(name of the variable + "=v"))
  end for
  initials ← "$c$.initial_state" + "{All values in initial_values, connected by /\}"
  print VARIABLES[variables, separated by commas]
  print MODES[modes, separated by commas]
  print STATESPACE + statespace, separated by linebreaks
  print INITIAL + initials, separated by line breaks
  print FLOW + flows, separated by line breaks
  print JUMP + jumps, separated by line breaks
  print UNSAFE
```

6 Conclusion and Future Work

In the introduction we have discussed that the present work deals with three subjects: hybrid systems, modeling languages and educational software (Figure 1.2). On the basis of these subjects, let us show the contribution made in this thesis.
In the field of hybrid systems, the specification language SoPHY contributed to the simulation of a certain class of hybrid systems, called hybrid automata. This language allows to develop tools, based on SoPHY, that simulate hybrid automata.
With the introduction of SoPHY, a new specification language for this class of hybrid systems was developed as a contribution to the subject of modeling languages. To develop the structure of this language, we considered the insides of scientific experiments, by identifying the characteristic parts of an experiments and modeled these into hybrid automata.
As a contribution to the third area, educational software, this thesis explores how scientific experiments can be better integrated into self-regulated learning. SoPHY is the preparation for a software tool in this context. Our design approach for SoPHY is the use in didactics, therefore several examples were used to elaborate in detail which properties can be found in almost all scientific experiments. By reference to this data, SoPHY was designed easy to readable and self-explanatory and determined that and how the experiments can be modeled with hybrid systems.

This thesis is a first step onto a tool that can be used in the school education as well as for the independent further education of the students at home. This tool is implemented in another thesis parallel to this work on the basis of the developed language [Yör17]. There are some points in this work that need to be discussed and examined in the future. In dialogues with didactics in the processing phase of this thesis, it has become clear that SoPHY is still to abstract to be used as the only input to such a tool.

In order to improve this fact, it would make sense to further simplify the syntax of SoPHY. If possible, a meta-level above SoPHY should be placed closer to the intuitive description of experiments. This could be achieved in finding a way to formulate an experiment by describing the dependencies of the objects that are used within the experiment.
In order to increase the performance of SoPHY, a possibility could be created of how to divide large systems into several smaller systems. This should be easy to implement because, only the syntax of the language must be expanded from individual systems, on lists of systems. The adaptation of the semantics and the investigation of the effects of parallel simulated systems could mean a greater effort. As a last point, the possibility of

verifying experiments could be integrated directly into SoPHY, so that no translation into other languages is necessary. The verification of the experiments would provide teachers with a possibility to check their modeling, as well as offer the students further learning experiences.

7 Bibliography

[ABRW17] A. Angermann, M. Beuschel, M. Rau, and U. Wohlfarth. MATLAB - Simulink - Stateflow: Grundlagen, Toolboxen, Beispiele. De Gruyter Studium. De Gruyter, 2017.

[ACH+95] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. THEORETICAL COMPUTER SCIENCE, 138:3–34, 1995.

[AD94] Rajeev Alur and David L. Dill. A Theory of Timed Automata. Theor. Comput. Sci., 126(2):183–235, apr 1994.

[AGH+00] Rajeev Alur, Radu Grosu, Yerang Hur, Vijay Kumar, and Insup Lee. Modular Specification of Hybrid Systems in Charon, pages 6–19. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.

[BCL+05] Christopher Brooks, Adam Cataldo, Edward A. Lee, J. Liu, Xiaojun Liu, Stephen Neuendorffer, and Haiyang Zheng. HyVisual: A Hybrid System Visual Modeler. Technical Report UCB/ERL M05/24, EECS Department, University of California, Berkeley, Jul 2005.

[BHR84] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A Theory of Communicating Sequential Processes. J. ACM, 31(3):560–599, jun 1984.

[BK08] Christel Baier and Joost-Pieter Katoen. Principles of Model Checking. The MIT Press, 2008.

[Bun] Bundesministerium für Bildung und Forschung. qualifizierung digital. https://www.qualifizierungdigital.de.

[Col81] L. Collatz. Differentialgleichungen: eine Einführung unter besonderer Berücksichtigung der Anwendungen. Leitfäden der angewandten Mathematik und Mechanik. Teubner, 1981.

[Cro06] Douglas Crockford. The application/json media type for javascript object notation (json). 2006.

[DGS00] Akash Deshpande, Aleks Göllü, and Luigi Semenzato. The SHIFT Programming Language and Run-time System for Dynamic Networks of Hybrid Automata, pages 355–371. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.

[EGK11] C. Eck, H. Garcke, and P. Knabner. Mathematische Modellierung. Springer-Lehrbuch. Springer Berlin Heidelberg, 2011.

[GNH13] Thomas Götz, Ulrike E. Nett, and Nathan C. Hall. Self-Regulated Learning. In Nathan C. Hall and Thomas Goetz, editors, Emotion, Motivation, and Self-Regulation : A Handbook for Teachers, pages 123–16. Emerald Group, Bradford, 2013.

[Gre12] Michael D. Greenberg. Ordinary Differential Equations. Wiley, 2012.

[Grö77] Wolfgang Gröbner. Differentialgleichungen : 1. Gewöhnliche Differentialgleichungen. Mathematik für Physiker. D. Laugwitz, P.Mittelstaedt, H.Rollnik, G.Süßmann, 1977.

[Grü08] Lars Grüne. Modellierung mit Differentialgleichungen - Vorlesungsskript. Universität Bayreuth, 2008.

[Hen] Martin Hennecke. Softwareentwicklung für den Mathematikunterricht. https://www.uni-hildesheim.de/fb4/institute/imai/mitglieder/dr-martin-hennecke/softwareentwicklung-fuer-den-mathematikunterricht-semu/.

[Hen96] Thomas A. Henzinger. The Theory of Hybrid Automata. pages 278–292. IEEE Computer Society Press, 1996.

[HHWt97] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-toi. HyTech: A Model Checker for Hybrid Systems. Software Tools for Technology Transfer, 1:460–463, 1997.

[HJ90] H. Hansson and B. Jonsson. A Calculus for Communicating Systems with Time and Probabilities. In [1990] Proceedings 11th Real-Time Systems Symposium, pages 278–287, Dec 1990.

[HKPV98] Thomas A. Henzinger, Peter W. Kopke, Anuj Puri, and Pravin Varaiya. What's decidable about hybrid automata? Journal of Computer and System Sciences, 57(1):94 – 124, 1998.

[HMP$^+$] Marit Hoch, Monique Meier, Felix Papsch, Lara Yörük, and Orcun Yörük. DiVoX. Master Project, University of Kassel.

[Hoh02] Markus Hohenwarter. GeoGebra – ein Softwaresystem für dynamische Geometrie und Algebra der Ebene. Master thesis, 2002.

[Hoh06] Markus Hohenwarter. GeoGebra – didaktische Materialien und Anwendungen für den Mathematikunterricht. PhD Thesis, 2006.

[Int] International GeoGebra Institute. GeoGebra. https://www.geogebra.org.

[Kle08] Wolfgang Kleier. Hybride Systems - Vortrag 2008. http://num.math.uni-bayreuth.de/de/teaching/archive/ss_2008/01069/Wolfgang_Kleier_Vortrag.pdf, 2008.

[LPY97] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. Int. Journal on Software Tools for Technology Transfer, 1(134–152), 1997.

[LTS08] John Lygeros, Claire Tomlin, and Shankar Sastry. Hybrid Systems: Modeling, Analysis and Control. December 2008.

[Mil80] Robin Milner. A Calculus of Communicating Systems, volume 94 of Lecture Notes in Computer Science. Springer-Verlag, New York, NY, 1980.

[MPW92] R. Milner, J. Parrow, and J. Walker. A Calculus of Mobile Processes, I and II. Information and Computation, 100(1):1–40,41–77, 1992.

[Pla10] Andre Platzer. Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics. Springer Publishing Company, Incorporated, 1st edition, 2010.

[Rat] Stefan Ratschan. RSolver User Manual. http://rsolver.sourceforge.net/documentation/manual.pdf.

[RDS] Stefan Ratschan, Tomáš Dzetkulič, and Zhikun She. HSolver User Manual. http://hsolver.sourceforge.net/documentation/manual.pdf.

[RS07] Stefan Ratschan and Zhikun She. Safety Verification of Hybrid Systems by Constraint Propagation-based Abstraction Refinement. ACM Trans. Embed. Comput. Syst., 6(1), feb 2007.

[Sta16] Dr Guy-Bart Stan. Modelling in Biology. Imperial College London, 2016.

[Yör17] Orcun Yörük. Ein Simulationstool für Hybride Systeme. Masterthesis, University of Kassel, 2017.