A Theory of Refinement of Cyber-Physical Systems into Implementations

Bachelor's Thesis of

Daniel H. Draper

at the Department of Informatics
Institute for Theorethical Informatics

Reviewer:	Prof. Dr. Bernhard Beckert
Second reviewer:	
Advisor:	Dr. rer. nat. Mattias Ulbrich

15. April 2015 – 15. August 2015

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, 14th of August, 2015

. . . . . .

(Daniel H. Draper)

Abstract

Cyber-Physical Systems or Hybrid Systems are increasing in importance everyday. Different approaches to analyzing and verifying both hybrid models in the form of hybrid programs or - automata are common practice, as well as approaches to analyzing and verifying discrete java programs. But a concrete way of gaining a concrete implementation from a hybrid model and verifying the complete system including both the hybrid model and implementation does not exist yet. In this thesis we close that gap between the two worlds. We propose and apply a way to refine hybrid systems in the form of hybrid programs into concrete java implementations and offer a formalized approach to verification of the entire systems using a third proof component(aside from the verification of the hybrid model using KeYmaera and the verification of the java control program using KeY), that we call the glue.

In the thesis we first showcase a motivating example of the control of a Watertank's water level, detailing the complex issue that is presented by hybrid system refinement and the issues we had with even the easy example we picked. Afterwards we present our approach to refinement of hybrid systems into implementations and verification of hybrid model/implementation packages in a formalized outline of steps to gain a verified implementation. At last we apply our newly found approach to a fresh example of a car and speed limit controller on a linear track progession ensuring the compliance of the car with the speed limit at all times.

Zusammenfassung

Cyber-Physical Systeme, für uns äquivalent zu Hybriden Systemen, werden heutzutage immer wichtiger. Verschiedene Ansätze um hybride Modelle in der Form von hybriden Programmen, als auch -automaten zu analysieren und verifizieren sind heute schon sehr üblich. Für Ansätze die diskrete Java Programme analysieren und verifizieren gilt dies genauso. Aber einen konkreten Ansatz um aus einem hybriden Modell eine konkrete Implementation zu erhalten existiert noch nicht. In dieser Arbeit schließen wir diese Lücke. Wir stellen einen Ansatz vor mit dem man Hybride Systeme gegeben in der Form von Hybriden Programmen in konkrete Java Implementationen refinen kann und wenden diesen an. Außerdem stellen wir einen formalisierten Ansatz vor um das komplette System bestehend aus dem Hybriden Modell und dem Java Kontrollprogramm zu verifizieren. Dafür führen wir einen dritten Beweisbaustein ein (Neben der Verifizierung des Hybriden Modelles und des Java Programmes mit KeYmaera und KeY resp.) den wir als “Glue” bezeichnen.

In dieser Arbeit stellen wir daher erst einmal ein motivierendes Beispiel in der Form eines Wassertanks, dessen Wasserlevel durch Anpassung des Ventils der Zuflussgeschwindigkeit in einem bestimmten Bereich gehalten werden soll vor. Das Beispiel zeigt die Komplexität die das Refinement eines Hybriden Systems mit sich bringt, selbst bei einem so einfachen Beispiel wie dem Wassertank. Danach stellen wir unseren Ansatz, wie man aus einem hybriden modells eine konkrete Implementation gewinnt und diese zusammen mit dem hybriden Modell verifiziert, konkret vor. Als letzes wenden wir unseren vorgestellten Ansatz dann auf ein frisches Beispiel an. In diesem wird die Beschleunigung eines Autos auf einer linearen Strecke kontrolliert und Geschwindigkeitslimits von einem weiteren Kontrollprogramm auf dieser Strecke verteilt. Das Auto und die Geschwindigkeitslimits sollen dann so kontrolliert werden, dass das Auto die Geschwindigkeitslimits zu jeder Zeit einhält.

Contents

List of Figures

List of Tables

1. Introduction

A growing number of automation today is done through computers and computer programs that reach outside of their digital world into the analog world, controlling some physical aspects. These systems are called cyber-physical systems (*CPS*) and are always safety-critical, due to the physical control they exert. While formal verification methods exist for hybrid models of CPS and formal verification methods exist for concrete implementations, a certain gap is evident in between the two, when trying to implement control programs for concrete CPSs.

In this bachelorthesis we formalize an approach attempting to close that gap: Replacing the abstract notion of the control program in a verified (by KeYmaera) Cyber-Physical-System (*CPS*) with a concrete, verified (by KeY) implementation through a form of Formal Refinement and being able to verify that the entire CPS still satisfies the required safety constraints, using KeYmaera.

CPS are generally modelled as either Hybrid Automata or - Programs.(See ref. [Pla10]). In these hybrid models, the control-part only exists in an implicit manner, while we require an explicit statement at which the control program is enacted, what we later on refer to as the “hook”.

To be able to verifiy the entire implementation against the cps-safety-constraint, we define a “glue”, which refers to a relation that translates real values into discrete values and vice-versa. In certain cases (See sect. 3.5), glue will be a concrete function and not only a relation, but not in general.

If we summarize the goal of this thesis we get 1. Here, glue is the aforementioned relation to be able to translate from the real into the discrete world.

1.	Verification of hybrid model against safety property α using KeYmaera.
2.	Verification of implementation using KeY.
3.	Verification of glue using KeYmaera.
$\Longrightarrow$	Hybrid model with concrete implementation inserted at hook fullfills safety property α.

This means: if we have a certain hybrid program with an explicit hook that models a CPS and fullfills safety condition α (meaning the CPS always terminates and is always in a state which fullfills α), an implementation that is correct and have found a valid, verified glue relation between the different values of the physical and discrete system-parts, then the hybrid program using the concrete implementation in place of the hook also fullfills safety condition α.

1.1. Related Work

Hybrid systems offer many challenges not present in discrete approaches to refinement, as the continuous evolution of reals means that computation is complex and state space as well as transitions can be uncountably infinite. Discrete refinement approaches (e.g for Z([CW98], [DB14]), or for Event-B([STW14])), do not really apply to the problems in the Cyber-Physical hybrid world. Timed automata (as presented in [AD94]), while closely related to hybrid systems as timed progression is one of the basic principles of hybrid systems, are still finite-state automata and do not solve the problem of abstracting a hybrid system with infinite states into a finite-state version.

Other refinement approaches as in [MV92] offer mathematical foundations for discrete refinement, but can not find application in our problem space. Counter-example guided refinement as presented in [CFH$^+$03], offers a way to abstract hybrid systems based on counter-examples found by a model checker to find a suitable finite-state abstraction for the hybrid system for easier verification. They does not present a way of refinement into a concrete implementation as we are trying to establish.

While refinement as a technique for analyzing hybrid systems in the context of implementations has been used in [GZ14], their approach offers only a refinement on a more abstract level, establishing a formalized approach to refinement of hybrid system and not the concrete implementation we achieve.

As we established a general procedure for finding an implementation, we found we needed an easy way to model a tick in our hybrid systems. One approach can be seen in [RyS96] using event-based automata as opposed to timed automata or infinite-state hybrid programs.

[RT13] takes a look at establishing a notion of robustness for Cyber-Physical-Systems, based on existing notions of robustness for discrete systems. It uses an abstraction and refinement process as to find robust versions of existing CPS models, but does not bother with actual implementations.

Overall we outline a more concrete approach to refinement as opposed to the more abstract presented approaches, resulting in real-world applicable control programs verified automatically using KeY and KeYmaera, as opposed to further abstractions of the problem as in the related works.

1.2. Outline

In this thesis we take a look at the following:

I: Preliminary Definitions.

II: Motivating Example: Refining a concrete case study: CPS “Watertank” (Example taken from [Pla15a]).

III: Introduction of approach to using Refinement to gain an implementation from a hybrid model.

IV: Application of newly found approach: case study: CPS “Traffic Control” (See ref. [MLP12]).

In the appendix, additional images are provided as well as the full source code for both hybrid programs and our implementations.

2. Preliminary Definitions

In the following chapter we want to define and terms, models and logic used throughout this thesis. We skip the explanation of first-order logic, which we presume to be known by the reader.

2.1. Introducing Cyber-Physical-Systems

In this thesis we take a close look at **CPS**. These are systems in which a physical aspect or value is controlled by a computer (program). For example, an aircraft control system in which the computer exerts a form of speed control on the airplane would be a CPS.

For our purposes we also refer to CPS as *Hybrid Systems*, in which discrete values (in the control program) and continuous values (in the physical world) coexist. The difficulty in analyzing these kinds of systems stems from the "hybridness" of the systems: There is always some form of translation necessary to go from the program (discrete values) to the physical (continous reals) world.

2.1.1. Modelling CPS as Hybrid Automata

Two different modelling approaches exist for CPS, the first one we now explain are hybrid automata. Formally they are defined to have the following components (From [Hen00]):

Variables A finite set $X = \{x_1, \ldots, x_n\}$ of real numbered variables. The number n is called the dimension of H. [...]

Control graph A finite directed multigraph (V, E). The vertices in V are called control modes. The edges in E are called control switches.

Initial, invariant, and flow conditions. Three vertex labeling functions $init$, inv and $flow$ that assign to each control mode $v \in V$ three predicates. [...]

Jump conditions An Edge labeling function jump that assigns to each control switch $e \in E$ a predicate.

Events A finite set Σ of events, and an edge labeling function event: $E \to \Sigma$ that assigns to each control switch an event.

Based on non-deterministic finite automata (NFA), they are easy to understand for humans and serve well to model the abstract behavior of hybrid systems. A possible hybrid automatacan be seen in Fig. 2.1. It describes the behavior of a controller controlling the train on a rail section. A train is therefore always either traveling freely (accelerating) or being slowed. The biggest difference to NFAs is apparent when looking inside both

states: They are continuous, so there is no single value assigned to the train's acceleration but rather a description of the behavior of the value over time is given. This means, that the single state "accelerate" can also be seen as a continous flowing assignment of values according to the specified differential equation. [Pla10].

The system still obviously exhibits its continuous behavior, as any change to the acceleration also brings the change of speed and location. As the train comes passes arbitrarily chosen point *SB*, the controller automatically assigns a discrete negative value to the train's acceleration, decelerating it, so it slows down. In the brake state the train again follows the equations of linear motion with continuous assignment, only that the guard $v \geq 0$ makes sure the train never travels backwards (with a negative velocity), but rather at some point starts accelerating again according to the discrete assignment in the state change between brake and accelerate.

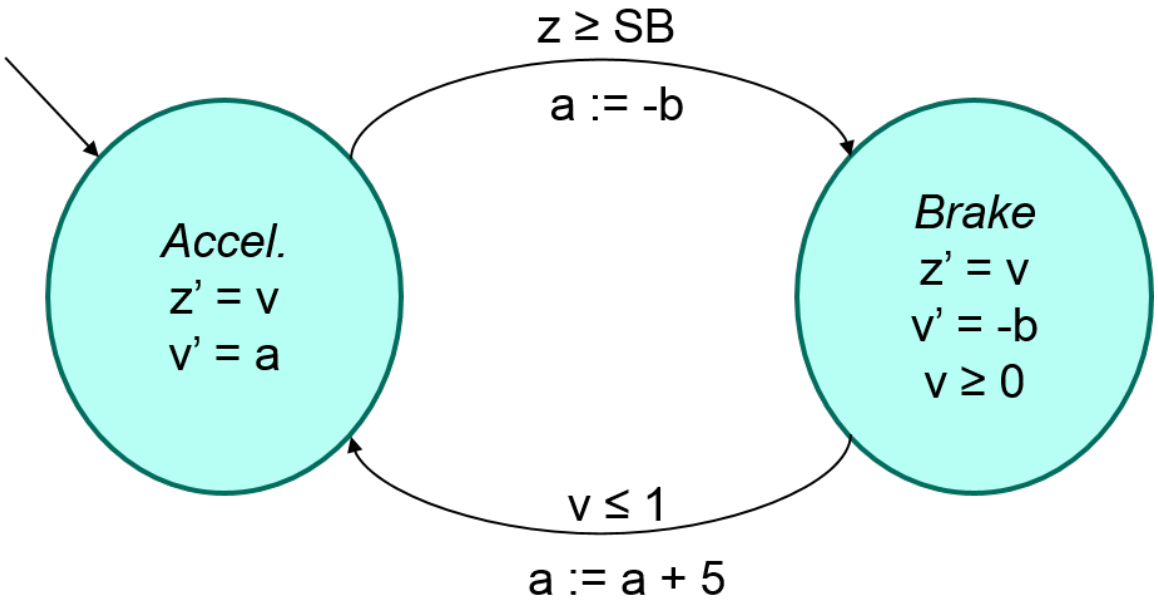

Figure 2.1.: A simple Hybrid Automata [Pla10].

2.1.2. Modelling CPS as Hybrid Programs

As we are using KeYmaera for verification, which uses Hybrid Programs (HP), we also have to introduce the syntax of hybrid programs. They follow the syntax depicted in table 2.1, whereby θ_i are terms, $x_i \in \Sigma$ are state variables and χ is a formula of first-order logic [Pla10].

As an example we take a look at the hybrid program notation of a train control sustem (See Fig. 2.2). As can be seen, the state variable x is first assigned the state accelerate, this corresponds to the hybrid automaton's (See Fig. 2.1) starting state of the system. In line 3 we can see the hybrid program syntax being an extension of first-order logic: Only if

notation	statement	effect
$x := \theta$	discrete assignment	assigns term θ to variable $x \in V$
$x := *$	nondet. assignment	assigns any real value to $x \in V$
$x'_1 = \theta_1 \wedge \ldots$ $\ldots \wedge x'_n = \theta_n \wedge \chi$	continuous evolution	diff. equations for $x_i \in V$ and terms θ_i, with formula χ as evolution domain
$?\chi$	state check	test formula χ at current state
$\alpha; \beta$	seq. composition	HP β starts after HP α finishes
$\alpha \cup \beta$	nondet. choice	choice between alternatives HP α or β
α^*	nondet. repetition	repeats HP α n-times for any $n \in \mathbb{N}$

Table 2.1.: Syntax of Hybrid Programs (HP) [Pla10].

$q = accel$ **and** $z \geq SB$ are true can this case be enacted (corresponding to the state change from accel. to brake in the automaton).

$q := accel;$
$((?q = accel; z' = v, v' = a)$
$\cup (?q = accel \wedge z \geq SB; a := -b, q := brake; ?v \geq 0)$
$\cup (?q = brake; z' = v, v' = -b \wedge v \geq 0)$
$\cup (?q = brake \wedge v \leq 1; a := a + 5; q := accel))^*$

Figure 2.2.: A simplified train control system as a Hybrid Program [Pla10].

2.2. Dynamic Logic

Dynamic logic is a multi-modal logic and has two basic operators (one of which is relevant to this thesis): Either *safety* ([]) in which a first-order-logic formula ϕ holds true in all exit states of the program α. Or *liveness* ($\langle\rangle$), where a possible execution of the program α exists, after which the first-order-logic formula ϕ holds. In this paper only the safety operator is used. They can be expressed as in Eq. 2.1, whereby α denotes the (hybrid) program that is enacted, and ϕ is the postcondition expressed in first-order-logic.

$$\begin{aligned} Safety &: [\alpha]\phi \\ Liveliness &: \langle\alpha\rangle\phi \equiv \neg([\alpha]\neg\phi) \end{aligned} \tag{2.1}$$

2.2.1. Differential Dynamic Logic

While dynamic logic in itself was not specifically devised for hybrid programs, this means we need a way to express differential equations, as most continuous movement or behavior is expressed by differential equations. DDL or d$\mathcal{L}$, the language also used by our verification tool KeYmaera, has ways to express differential equations as can be seen in Table 2.1.

2.2.2. JML: Verification of Java Programs using KeY

In this thesis we also deal with the verification of Java programs (the control programs of our cps). We now want to introduce the Java Modelling Language *JML*, which is used to model specification of Java classes/modules or single methods. JML is based on the Design-By-Contract premise, which means that the user and the Java classes have a contract with each other which defines the input and output conditions of the program [BHS07]. JML is then listed as a comment in the Java sourcecode right before the method in question [ABH+07]. For example the JML code for a method in question could look like this:

```
1
2 /*@ public normal_behavior
3    @ requires a >= 0
4    @ ensures b >= 2
5    */
6   public int getPlusTwo(int a) {
7    b = a + 2;
8    return b;
9 }
```

As one can see, the JML proof obligation needs to have the access level and a name and (for our purposes) always will have a *requires* precondition that must hold true before the method is called for all arguments or some class variables and a *ensures* postcondition clause, that has to hold true for all exit states of the method (where the precondition was true before calling it). This means the JML above could also be expressed in Java Dynamic Logic as:

$$a \geq 0 \implies [b := getPlusTwo(a)]b \geq 2 \tag{2.2}$$

2.3. KeYmaera as our Verification Tool for Hybrid Programs

As KeY can only deal with Java programs, but we deal with problems with hybrid systems, we use KeYmaera as our tool of choice to verify our hybrid models. It is based on KeY's source and developed by André Platzer at Carnegie Mellon University [Pla15b]. As its backend arithmetic solver we used Wolfram Mathematica [Wol], but KeYmaera supports other computer algebra solvers as well.

Hybrid Programs are expressed as *regular programs*, but as for example $\cup$ is not a symbol used in ASCII text, the program notation in the *.key* KeYmaera-files is different (e.g. $\cup$ as ++), but we will present every hybrid program in the syntax presented above as well as to avoid confusion and adding another layer of abstraction.

3. Motivating Example: Refinement of CPS "Watertank"

To better understand the process of refinement of a CPS we take an example from the KeYmaera tutorial [Pla15b], a watertank (See Fig. 3.1) whose water level is adjusted automatically to stay at a certain level, and refine its hybrid model to gain an implementation. Our goal therefore is finding a suitable and verified implementation from the hybrid model with all necessary proof steps. To accomplish this, we have to adjust and verify the hybrid model, code and verify a concrete implementation and finally verify the "glue" in between the discrete and continuous world we come up with as our last proof steep.

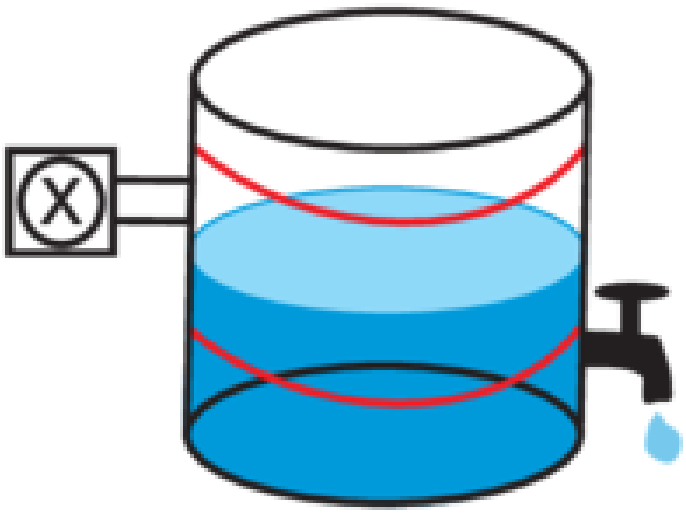

Figure 3.1.: Picture of possible Watertank configuration: Watertank that can get drained at all times and has a valve controlled by a control program [Pla15a].

3.1. Detailing the Watertank CPS

The "control" part of this hybrid system is the valve, it can either drain the watertank with a rate of -2 or it can fill the tank further with a rate of 1. The goal of the entire system then is, to keep the water level y of the tank between 1 and 12. Normally this is proven by KeYmaera without actually implementing a control program. To be able to we first took a look at the hybrid model of the watertank. It is provided both in the form of a hybrid automaton (See Fig. 3.2) and a hybrid program (See Sec. 3.1.1). In the hybrid automaton, the system starts in the "fill" state, where water is constantly flowing into the tank with

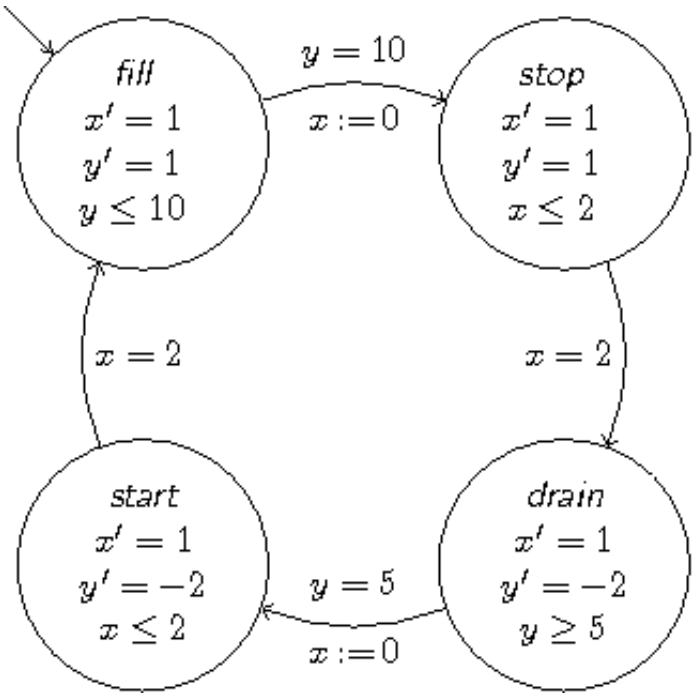

Figure 3.2.: The Watertank CPS expressed as a hybrid automata [Pla15a].

the rate of $y' = 1$, while the clock is counting by $x' = 1$ per tick. Once the water level reaches a certain level $y = 10$ the clock is reset to $x := 0$ and the state changes from "fill" to "stop", where the water level is still raising for 2 ticks since the valve can not change in instant time. Once those 2 ticks pass $x = 2$ the watertank switches its state again to "drain" in which water flows out of the tank at the rate of 2 per clock tick, before it changes again to "start" the filling of the tank agan at the water level of $y = 5$.

3.1.1. The original Hybrid Program

The hybrid program (see App. A.5 for HP expressed in ASCII) has three variables (water level y, time x and state st all as Real numbers) defined as: $\mathbb{R}\ y, x, st$; in the *ProgramVariables* section. Then the complete program notation with a preceeding ini-

tialization of values and state.

$$
\begin{aligned}
[x := 0, y := 1, st := 0]((st = 0) \implies & \\
& [(?(st = 0); \\
& \quad (?(y = 10); x := 0; st := 1) \\
& \quad \cup (?(y < 10 \lor y > 10); x' = 1 \land y' = 1 \land y \leq 10)) \\
& \cup (?(st = 1); \\
& \quad (?(x = 2); st := 2) \\
& \quad \cup (?(x < 2 \lor x > 2); x' = 1, y' = 1 \land x \leq 2)) \\
& \cup (?(st = 2); \\
& \quad (?(y = 5); x := 0; st := 3) \\
& \quad \cup (y > 5 \lor y < 5); x' = 1, y' = -2 \land y \geq 5)) \\
& \cup (?(st = 3); \\
& \quad (?(x = 2); st := 0) \\
& \quad \cup [(?(x > 2 \lor x < 2); x' = 1, y' = -2 \land x \leq 2)) \\
&](y \geq 1 \land y \leq 12))
\end{aligned}
$$

3.2. The "Hook": Concrete Control Value Assisgnment

In order to be able to apply Eq. 1 to this concrete example, we first found a spot in which a (or multiple) concrete control value is actually assigned. We call this assignment(s) the *hook* as the actual implementation will "hook" into our hybrid model at this exact point. The Hybrid Automata describing the Watertank (See Fig. 3.2), required changing to be able to find our actual "hook". In the original model, a control value is never explicitly assigned, rather does the valve change its state non-deterministically, making a deterministic control program implementation impossible, as the program can not act in a non-determnistic manner and would not fit in the model of the cps. A version of the hyrbrid automata with the hook marked in red can be found at App. A.1. Our remodelled system features a clear ticked hook that is called upon at deterministic times (See Sec. 3.2.1).

3.2.1. Remodelled Hybrid Program

The remodelled hybrid program(see App. A.5 for HP expressed in ASCII) has 5 variables and a constant: tick defined in the
$functions$ clause as being $\in \mathbb{R}$, and water level y, time x, new valve value new and valve value from a tick ago valve which are defined in the

programVariables clause as being $\in \mathbb{R}$.

$$
\begin{aligned}
&tick > 2 \implies \\
&\quad [x := tick, y := 1, valve := 1; \\
&\quad (?(x = tick); new := *; x := 0); \\
&\quad\quad (?(y + 2 * valve + tick * new \geq 1 \land y + 2 * valve + tick * new \leq 12 \land \\
&\quad\quad (new = 1 \lor new = 2)); \\
&\quad\quad\quad ((?(new \neq valve); x' = 1, y' = valve \land x \leq 2); \\
&\quad\quad\quad\quad (\text{if } (x = 2) \\
&\quad\quad\quad\quad \text{then } valve := new; x' = 1, y' = valve \land x \leq tick \text{ fi}) \\
&\quad\quad\quad (?(\neg(new \neq valve)); x' = 1, y' = valve \land x \leq tick)))] \\
&\quad (y \geq 1 \land y \leq 12)
\end{aligned}
$$

We kept the model mostly equivalent to the original, preserving the two clock ticks of time needed for the valve to change its state from drain to fill and vice-versa. This presented a challenge, when thinking of possible implementations, as the program hook could be called again before the valve would have actually changed its state. To simplify this problem,we only model cases in which the actual program tick (so the time between two hook calls) is greater than two clock ticks (however long they may be). This means, that the valve or control program does not have to have a (possibly infinite) list of the last control values assigned, which could be the case if the program was called more frequently than the valve is able to change states.

3.3. Implementation Safety Condition

Next we devised a postcondition for the implementation hook, which would then serve as the abstraction of the java control program. This means, that the program would be built accordingly, so that it could be verified against this postcondition. It therefore also served as a form of specification for us when devising the implementation. The original postcondition we devised:

$$
\begin{aligned}
\psi \equiv (y + 2 * valve + tick - 2 * new \geq 1 \land \\
2 * valve * (tick - 2) * new \leq 12 \land \\
y + 2 * valve \geq 1 \land y + 2 * valve \leq 12)
\end{aligned}
\tag{3.1}
$$

During verification of the entire hybrid program, KeYmaerawould not verify the hybrid program with this postcondition. This means, that even in such a simple CPS as this watertank control system, finding the hook postcondition was non-trivial and only with the help of KeYmaera's counterexamples did we manage to find the correct postcondition (See Eq. 3.2).

$$\psi \equiv (y + 2 * valve + tick * new \geq 1 \wedge \\ y + 2 * valve + tick * new \leq 12 \wedge \\ (new = 1 \vee new = -2)) \tag{3.2}$$

KeYmaera verified our new hybrid program fully automatically.

3.4. The (simple) Java Control Program

After completing verification of our hybrid model with KeYmaera, we proceeded to implement a (simple) control program for the Watertank. Finding a suitable implementation proved difficult, as a parallel implementation (Both the control system and the differential equations working in parallel, modifying the water level) seemed more intuitive. This would defeat the purpose of actually finding a suitable hook and was more based on our understanding of the original hybrid model and not our version including the hook.

Using our understanding of a singular hook of the control system into the watertank as well as the hook postcondition from the previous section as our specification for the actual control method, that would return the control value to the watertank's valve, we then managed to implement a suitable discrete control method (See Lst. 3.4). As can be seen, the control method is relatively simply and only assigns the best/worst case scenario values. For the entire control class see App. A.2.

```
1  public int getControlValue (int y, int old) {
2        //Waterlevel in two time units
3        int inTwo = y + 2 * old;
4        //If we are raising level, keep raising if possible without
             hitting max_level before next tick
5        if (old == 10) {
6           if (inTwo + tick * 1 <= 116) {
7              return 10;
8           }
9           else {
10                return -20;
11          }
12       }
13       //ELSE if we are currently lowering level, keep lowering if we can
             lower further without hitting min_level b4 next tick
14       else {
15          if (old == -20) {
16             if (inTwo - tick * 2 >= 12) {
17                return -20;
18             }
19             else {
20                   return 10;
21
22             }
23
24          }
25       }
```

```
26        //Only returned if old != 10 && old != -20, unreachable.
27        return 0;
28    }
29    }
```

The control method consists only of simple if-else-statements and just assigns the best-case (keep raising level if we were raising, keep lowering if we are lowering as long as postcondition is still valid) value to the valve. We translated the hook postcondition (See Eq. 3.1) into, what we thought to be, a suitable JDL postcondition statement for verification with KeY (See List. 3.4). In the next section we will explain why this postcondition would not work for our complete proof. KeY verified our program fully automatically, aside from the fact, that the $tick > 2$ precondition added too much computational complexity, so we let KeY prove the property with the more concrete $tick == 30$; $tick == 31$;... assignments.

For testing purposes, we also implemented a complete simulator of the Watertank CPS (See App. A.3).

```
1     /*@ public normal_behavior
2       @ requires tick == 30 && y >= 10 && y <= 120 && y + 2 * old <= 120
            && y + 2 * old >= 10 && (old == 10 || old == -20);
3       @ ensures \result * (tick / 10) + y + 2 * old  >= 12 & \result * (
            tick / 10) + y + 2 * old <= 116 && (\result == 10 || \result ==
             -20);
4       @*/
```

3.5. Glue between Java and Hybrid Model

As real values (used in the hybrid model) and discrete values (used in our java implementation) are fundamentally different, a certain transformation has to occur. In the Watertank example, this means, that the waterlevel as well as the last valve-value (be it fill or drain), which are passed to the control method, have to be converted into one direction by the glue.

Also, the result of the computation in the method (so the new valve value) has to be converted in the other direction. In general the glue is not necessarily a bijective function, as some values will not exist in both worlds, making it a relation. For the Watertank, fortunately, this problem doe not exist, so finding a function translating the waterlevel and valve values into each other instead of a relation simplifies the problem.

Finding the glue between the real parts of the hybrid system and our java control program still proved difficult though.

As we went along, we figured out how a general glue proof should look (See Fig. 3.3). Here x are the discrete values in our control program, y are the real world values, ψ is the postcondition of the control program expressed using discrete values and ϕ is the actual safety condition we want to show for the entire hybrid system.

$$\forall\, x \in (DiscreteWorld)\ \forall\, y \in (RealWorld) : (x, y) \in glue \implies (\psi(x) \implies \phi(y)) \quad (3.3)$$

Following this basic guideline, we were able to find a suitable glue relation (or in our case total relatin) for this cps. We found the following as the glue-relation:

$$\begin{aligned}\forall\, y \in \mathbb{R}\,.\forall\, y_j \in \mathbb{Z}.\forall\, valve \in \mathbb{R}.\forall\, tick \in \mathbb{R}.\forall\, tick_j \in \mathbb{Z}.\forall\, new \in \mathbb{Z}.\forall\, result \in \mathbb{R}\ :\\ y_j = \text{floor}(10 * y) \wedge old_j = \text{floor}(10 * valve) \wedge tick_j = \text{floor}(10 * tick) \wedge\\ result = 10 * new \wedge y \geq 1\ \wedge\ y \leq 12\ \wedge (valve = 1\ \vee\ valve = -2) \wedge\\ tick > 2\end{aligned} \tag{3.4}$$

Whereby $\psi(x)$ is equivalent to:

$$\begin{aligned}result * tick_j/10 + y_j + 2 * old_j \leq 116 \wedge result * tick_j/10 + y_j + 2 * old_j \geq 12 \wedge\\ (result = 10 \vee result = -20)\end{aligned} \tag{3.5}$$

And $\phi(y)$ is equivalent to:

$$\begin{aligned}y + 2 * valve + tick * new \geq 1 \wedge y + 2 * valve + tick * new \leq 12 \wedge\\ (new = 1 \vee new = -2)\end{aligned} \tag{3.6}$$

As can be seen, the $\psi(x)$ that we found for the glue is not equal to the original trivial postcondition for our java program (See List. 3.4). This can be explained by the floor functions that has to be used for translation from the $\mathbb{R}$ into $\mathbb{Z}$ world. Due to the usage of the floor function, we only can approximate the original value in the discrete world, and therefore these approximation erorrs have to be accounted for in ψ. Again the fact that we only found this error in our original postcondition devised for the java program, proves the non-triviality of the process even in seemingly simple examples as this watertank. The glue in ASCII notation can be found in App. Sec. A.5.

3.6. Verification based on KeYmaera

Verification of the glue proved difficult since KeYmaera, or specifically Mathematica (cf. [Wol]) (KeYmaera backend algebraic solver), has problems with the translation of unintepreted functions. Therefore we found an abstraction of the floor function using inequalities to approximate it (See Eq. **??**).

$$\forall\, x \in \mathbb{R}.\forall\, c \in \mathbb{R}.floor(x) = c\ \Longrightarrow\ x - 1 < c \wedge c \leq x \wedge (x > 0\ \Longrightarrow\ c > 0)$$

Also, since KeYmaera does not deal with numbers outside of the reals, we generalized the preconditions for every variable to be in the reals. This obviously only strengthens the proof and therefore does not change the correctness.

This still lead to problems when using the automatic proof-mechanism, so we had to add a full new taclet fdef to the definition file to be able to use the automatic proover (See App.. A.2).

The proof runs fully automatically with our own defined taclet.

4. Formalized approach of refining hybrid models into implementation

What the Watertank example shows is the non-triviality of refining the hybrid model into an implementation and of the verification of all necessary parts. Overall it is obvious, that a formalized approach to the general problem presented in chapter 1 is necessary. In this chapter we present a possible formalized approach to the problem, that we deemed feasible.

To aid readability we will now give an overview of the process without explanation, then detailing each step in the following sections.

1. Modelling CPS as Hybrid System that includes concrete hook.

2. Finding the necessary safety condition of the control value for verification with KeYmaera.

3. Implementing control program according to (preiliminary) safety condition and Verification by KeY.

4. Finding the correct “glue” between hybrid model and control program and its verification by KeYmaera.

5. Result validation: Does our implementation still match the correct safety condition after verifying the glue?

Overall for verification of our resulting implementation three proof obligations exist:

i Proving correctness of our hybrid model that includes a hook and (probably) the hook postcondition with KeYmaera.

ii Proving correctness of our implementation according to the (preliminary) safety condition and verifiying it against it using KeY.

iii Proving correctness of our glue using KeYmaera.

With correct verification of all three parts, we can deduce that Eq. 1 has been fullfilled and our goal has been reached.

4.1. Modelling Hybrid System with Implementation Hook

Most CPS we took a look at (See [Pla15b] Tutorial, [Pla10, p. 5, p. 11] ...) as examples, did not have a concrete spot in which a control program could "hook" in easily. This means, that the first step in our refinement process has to be finding a suitable hook for the control program, referring to one or more non-deterministic assignments of a/multiple control values. Mostly for already existing hybrid systems, as was the case in Sec. 3.2, a complete changing of the model is necessary. When creating a new model, one should try to find a suitable spot for a non-deterministic control value assignment. As a discrete deterministic control program can only be enacted at certain times, a model of a form of tick has to be found, as to make an actual implementation feasible. Mostly this will result in an extra condition for the entire hybrid program, as we only want the program to work if the differential equations run till the discrete tick(See Sec. **??**).

To prevent issues in later verification of the model, a hybrid program and not a hybrid automata representation of the model should be chosen (cf. [Pla10, ch. 1.1.4]). Since the modelling of the system can often be incorrect and a formal way to verify if the model correctly models all necessities does not exist, great care should be enacted during modelling of the cps.

4.2. Devising Control Value Safety Postcondition

After finding a suitable hook spot in our program we have the analyze the hook for a safety condition. If we think back to the watertank example (See ch. **??**), this was non-trivial as the first postcondition we came up with through logic deduction proved to be incorrect. When thinking of possible conditions we often found we overlooked the tick when considering different postconditions. Obviously, more than one valid postcondition exists, as the constraints imposed on the implementation can get arbirtrarily strong and we're not restricting the actual behavior of the physical part of the cps.

One should try and find the most general postcondition ψ for which the safety condition still holds, as this makes it easier to code the actual implementation with less constraints. Of course, if the model's proof works without any postcondition, this means that the program could also just assign random values making the implementation and glue trivial or non-existant. With correct modelling and sufficiently complex hybrid system (which even the relatively easy watertank had) this should never occur though.

The entire hybrid model with the hook safety postcondition should then be verified by KeYmaera.

4.3. Implementation according to Postcondition and its Verification

Implementation of the program can theorethically be done in any preferred language, for usage with KeY as the automatic prover, Java is preferred. The implementation is free from any constraints asides from the postcondition that was devised previously, and from the

constraints provided by KeY itself (No threading etc.). Since the postcondition is in the hybrid model and therefore written in the form of real variables and values, one has to already think of a form of the glue that will be formally devised as the next step, as to find a suitable translation of the postcondition expressed with real values into JDL. This can be a preliminary condition, as it is possible (as in Sec. 3.5) that the formal glue will show errors in the original safety condition.

The implementation with its postcondition expressed in JDL then has to be verified by KeY before continuing with the glue.

4.4. Glue between Hybrid Model and Implementation and its Verification

As both our model and implementation are now verified, the actual glue between the implementation and the model has to be devised. In the previous Chapter we came up with a general definition of what the glue relation provides:

$$\forall\, x \in (DiscreteWorld)\ \forall\, y \in (RealWorld) : (x, y) \in glue \implies (\psi(x) \implies \phi(y)) \quad (3.3)$$

Coming up with the glue is the hardest part of the entire process, and took multiple iterations before a correct and verified glue was found. Most of the time the glue will probably incorporate some form of rounding/flooring mechanism, as we have to translate real values into discrete ones. This makes the entire proof hard to fully understand for humans without lots of complex mathematical computations.

We used a form of logical deduction to come up with the correct quantifiers for the glue proof (so the $\forall$ in Eq. 3.3), which can be found in App. **??**. Verification of the glue should then happen using KeYmaera, which means formulation of the glue should already be in the required syntax of DDL.

Overall this means, that for the discrete-world postcondition ψ that was proven by KeY , the real-world postcondition ϕ found in sect. 4.2, and the glue that we devise which is verified by KeYmaera, we reach the goal we set out for (See Ch. 1).

4.5. Validating Verificiation results against Implementation

As we realized in our Watertank example (See Sec. 3.5), with verification of our glue some changes might have to be made to the implementation and its postcondition. To complete the process of refinement we have to change the implementation and JDL condition and verify it again with KeY , make sure all our decisions made were actually valid and correct when taking a look back at the entire picture as outlined in Ch. 1.

5. Evaluation using Example CPS: "Traffic Speed Control"

After formalizing our approach used in the opening watertank example, we now want to prove its feasability on a new and fresh example. The following example is taken from [MLP12] and describes a two-part control system responsible for a one-way linear freeway on which speed limits are set randomly by one part of the control system and the one moving car's acceleration is set by the car's part of the program. The goal for the system is to never let the car break the speed limit at any point. This means the two control systems have to communicate, which also has to be represented in the hybrid system.

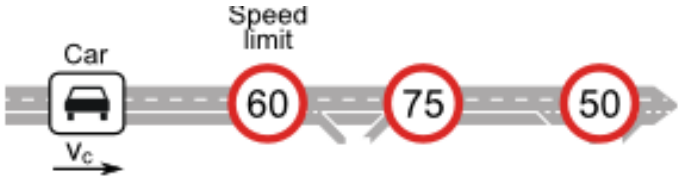

Figure 5.1.: The set-up of this CPS [Pla15b].

To show feasibility of our newly found approach, we now follow the process outline from the previous chapter. The first step is remodelling the CPS as to include a hook and a suitable safety condition and verifying it, followed by implementation and verification of the entire system.

5.1. Remodelling with Hook

When taking a look at the original hybrid program notation of thes traffic control cps (See Eq. 5.1.1), one can see the complexity of this problem. Here, t is the time, x_1 is the position of the car, x_{sl} is the position of the next speedlimit, v_1 is the speed of the car, v_{sl} is the speed limit, A is the maximum acceleration, B is the maximum braking power and ep is the maximum communication time between the car and speed limit controller. This hybrid program notation already obviously brings a high level of complexity and is hard to understand, so we decided to simplify the problem during remodelling.

5.1.1. Remodelled Hybrid Program

To simplify the problem at hand, we first removed the communication time ep, by arguing, that it can be 0 without affecting the correctness of the model, but for our purposes using

ep as the length of the tick, as this is the time in which we cannot exert control over the system. To introduce the tick we use the same trick as in the watertank example by only "executing" the differential equations when the tick was reached and forcing them to run the full duration until the next tick every time. Also, for simplification we tick the speed limit control system synchronized to the car control system, as to remove complexity and make the model easier. This seems plausible in reality and we forego the need to model/implement the communication between both control programs. Our remodelled hybrid program can be seen in Eq. 5.1.1. Here, the two hooks can be seen at $a_1 := *;$ and $x_{sl} := *; v_{sl} := *;$.

In our remodelled version, the car is controlled at an arbitrary time $t \ mod \ ep \equiv 0$, whose control program then ensures safety for the car until the next tick occurs and the program is enacted again, meaning if the car breaks at maximum braking speed$B$ at the next tick it can still reach a safe position/speed. Then at the same time t the second hook for the speed limit control program is enacted, where another postcondition has to be checked as to make sure the new chosen speed limit is in a safe spot for the car.

$$
\begin{aligned}
&v_1 \geq 0 \land v_{sl} \geq 0 \land x_1 \leq x_{sl} \\
&\quad \land\ 2 * B(x_{sl} - x_1) \geq v_1^2 - v_{sl}^2 \\
&\quad \land\ A \geq 0 \\
&\quad \land\ B \geq 0 \\
&\quad \land\ ep \geq 0 \implies \\
&\quad [a_1 := -B \\
&\qquad \cup (?(x_{sl} \geq x_1 + (v_1^2 - v_{sl}^2)/(2 * B) + (A/B + 1) * (A/2 * ep^2 + ep * v_1)); \\
&\qquad\quad a_1 := *; \\
&\qquad\quad ?(-B \leq a_1 \land a_1 \leq A)) \\
&\qquad \cup (?(x_1 \geq x_{sl}); \\
&\qquad\quad a_1 := *; \\
&\qquad\quad ?(-B \leq a_1 \land a_1 \leq A \land a_1 \leq (v_1 - v_{sl})/ep)); \\
&\qquad (x_{sl} := x_{sl}; \\
&\qquad\quad v_{sl} := v_{sl}) \\
&\qquad \cup (x_{sl} := *; \\
&\qquad\quad v_{sl} := *; \\
&\qquad\quad ?(v_{sl} \geq 0 \land x_{sl} \geq x_1 + (v_1^2 - v_{sl}^2)/(2 * B) + (A/B + 1) * (A/2 * ep^2 + ep * v_1))); \\
&\qquad t := 0; \\
&\qquad \{a_1' = v_1, v_1' = a_1, \\
&\qquad t' = 1, v_1 \geq 0, t \leq ep\}* \\
&\quad](x_1 \geq x_{sl} \implies v_1 \leq v_{sl})
\end{aligned}
$$

$$
\begin{aligned}
&v_1 \geq 0 \land v_{sl} \geq 0 \land x_1 \leq x_{sl} \\
&\quad \land\, 2 * B(x_{sl} - x_1) \geq v_1^2 - v_{sl}^2 \\
&\quad \land\, A \geq 0 \\
&\quad \land\, B \geq 0 \\
&\quad \land\, ep \geq 0 \implies \\
&\quad [(\\
&\qquad ?(t = ep); \\
&\qquad (a_1 := *); \\
&\qquad ?(-B \leq a_1 \land a_1 \leq A \land (x_1 \geq x_{sl} \implies (a_1 \leq (vsl - v1)/ep)) \\
&\qquad \land\, (x_1 < x_{sl} \implies (x_{sl} \geq x_1 + (v_1^2 - v_{sl}^2)/(2 * B) + (A/B + 1) * (A/2 * ep^2 + ep * v_1)))); \\
&\qquad x_{sl} := *; v_{sl} := *; \\
&\qquad\quad ?(v_{sl} \geq 0 \land v_{sl} < v_1 \implies \\
&\qquad\quad x_{sl} \geq x_1 + (v_1^2 - v_{sl}^2)/(2 * B) + (A/B + 1) * (A/2 * ep^2 + ep * v_1)) \\
&\qquad\quad \land\, v_{sl} \geq v_1 \implies a_1 \leq (v_{sl} - v_1)/ep)); \\
&\qquad t := 0; \\
&\qquad \{x_1' = v_1, v_1' = a_1, \\
&\qquad t' = 1, v_1 \geq 0, t \leq ep\}) * \\
&\quad](x_1 \geq x_{sl} \implies v_1 \leq v_{sl})
\end{aligned}
$$

5.2. Safety Hook Postconditions

Following our process outline, the next step is to find a suitable safety postcondition for the two hooks. Using logic deduction we came up with two suitable safety postconditions (See Eq. 5.1, 5.2), as KeYmaera then automatically verifies the entire hybrid model. In the first postcondition, meant for the car control program, the chosen acceleration for the car has to be in between the minimum and maximum acceleration and the acceleration has to be chosen in such a way, that both cases are safe: 1) the car is currently before the speed limit, we accelerate in a way so that we can still break with the maximum braking speed at the next tick and stay under the limit or to the right of the speed limit. 2) If the car is already in the speed limited zone we accelerate in such a way that we don't reach a speed higher than the speed limit until the next tick. The second postcondition, responsible for the speed limit chooser program, has to make sure that the new speed limit is bigger than 0 (so as to make it possible for the car to even keep the speed limit), and that the speed limit is chosen in such a way that the car can still break with the maximum brake speed and be safe.

$$
\begin{aligned}
\phi_1 \equiv &- B \leq a_1 \land a_1 \leq A \land (x_1 \geq x_{sl} \implies (a_1 \leq (vsl - v1)/ep)) \\
&\land\, (x_1 < x_{sl} \implies (x_{sl} \geq x_1 + (v_1^2 - v_{sl}^2)/(2 * B) + (A/B + 1) * (A/2 * ep^2 + ep * v_1)))
\end{aligned} \tag{5.1}
$$

$$\phi_2 \equiv (v_{sl} \geq 0 \wedge v_{sl} < v_1 \implies x_{sl} \geq x_1 + (v_1^2 - v_{sl}^2)/(2 * B) + (A/B + 1) * (A/2 * ep^2 + ep * v_1)) \\ \wedge v_{sl} \geq v_1 \implies a_1 \leq (v_{sl} - v_1)/ep)) \tag{5.2}$$

5.3. Implementation according to postconditions

For the implementation of the two control programs we chose to use a conservative, easier to understand approach, as KeY already had problems when trying to verify a more complicated approach. As there are two control programs/methods in use here, we just established the necessary "communiction" betwen both through our third Simulator class (See App. A.4).

5.3.1. The Car Control Program

For the car control program (See List. 5.3.1), one can see, that we only change the acceleration value by one if it fullfills the safety condition and break the maximum amount if not. This means, that more often than not we will be breaking at full speed, and this control program could be improved greatly in future work.

```
1   public int control(int y, int v, int accel, int sl, int slPos) {
2
3       if (y >= slPos) {
4           if ((accel+1)*TICK <= (sl-v)) {
5               if (accel+1 <= MAXACCEL) {
6                   return accel+1;
7               }
8
9           }
10          else {
11              if((accel-1)*TICK <= (sl-v)) {
12                  if (accel-1 >= MAXBREAK) {
13                      return accel-1;
14                  }
15          }
16          }
17      }
18      else {
19          if (2*slPos*MAXBREAK*(-1) >= y*2*(-1)*MAXBREAK + (v*v - sl*sl)
                + ((accel+1)+(-1)*MAXBREAK) * ((accel+1) * TICK*TICK + 2*
              TICK * v)) {
20              if (accel+1 <= MAXACCEL) {
21                  return accel+1;
22              }
23          }
24          else {
25              if (2*slPos*MAXBREAK*(-1) >= y*2*(-1)*MAXBREAK + (v*v - sl*
                  sl) + ((accel-1)+(-1)*MAXBREAK) * ((accel-1) * TICK*
                  TICK + 2*TICK * v)) {
```

```
26                  if (accel-1 >= MAXBREAK) {
27                      return accel-1;
28                  }
29              }
30          }
31      }
32          return MAXBREAK;
33
34  }
```

In the contract we define for the car control program (See List. 5.3.1), one can see we had to explicitly list the value of the TICK and MAXBREAK/MAXACCEL as to make the automatic verification possible. We also had to give both the invariant as well as safety condition of the hybrid program as a precondition for the method, as we also did in our first watertank example (See List. 3.4), the first two preconditions can be seen as an invariant of the method, as we cannot control the acceleration level in our limit, if the acceleration level we get already violates this safety condition.

```
1 /*@ public normal_behavior
2     @ requires MAXBREAK <= accel && accel <= MAXACCEL && ((!(y >= slPos
          )) || v <= sl) && v >= 0 && sl >= 0 && TICK == 1 && MAXBREAK ==
           -10 && MAXACCEL == 10 && 2 * MAXBREAK * (slPos - y) >= v*v -
          sl*sl && (v <= sl || slPos * 2 * MAXBREAK >= y*2*MAXBREAK + (v*
          v - sl*sl));
3     @ ensures MAXBREAK <= \result && \result <= MAXACCEL && ((!(y >=
          slPos)) || (\result*TICK <= (sl - v))) && ((!(y < slPos)) ||
          (2*slPos*MAXBREAK*(-1) >= y*2*(-1)*MAXBREAK + (v*v - sl*sl) +
          (\result+(-1)*MAXBREAK) * (\result * TICK*TICK + 2*TICK * v)));
4     @*/
```

5.3.2. The Speed Limit Control Program

List. 5.3.2 shows the source code of the control method of our Speed Limit controller. The control method implements the two control values it has to assign by creating an array and returning the results in the array. The implementation proved interesting, as it has a certain freedom in choosing the next speed limit, as the control method could be trivial in case of just always returning the current speed limit and position, then the car would just continue with exactly it's current movement until it has to break and then come to a full stop. The control method tries to raise the current speed limit by one if we currently have a low speed limit set and lower it if we have a high speed limit set. We then assign the position of the speed limit as great enough to still be correct according to the set safety condition. Again this control method errs on the sude of caution and does not have a mathematically very interesting way to assign new values, something that could be improved upon in future work. The complete source code for the control class can be found in App. A.2.

```
1 public int[] control(int y, int v, int accel, int sl, int slPos) {
2       int[] result = new int[2];
3       result[0] = sl;
4       result[1] = slPos;
```

```
5       if (sl<10) {
6           if (sl+1 < v) {
7               result[0]++;
8               result[1] = y + (v*v - result[0]) + (MAXACCEL  + 1) * (
                    MAXACCEL * TICK*TICK + 2 * TICK * v);
9           }
10          else {
11              if (accel*TICK <= result[0]+1 - v) {
12                  result[0]++;
13              }
14          }
15      }
16      else {
17      if (result[0]-1 < v) {
18          if (result[0]-1 >= 0) {
19          result[0]--;
20
21          result[1] = y + (v*v - result[0]) + (MAXACCEL  + 1) * (MAXACCEL
                * TICK*TICK + 2 * TICK * v);
22          }
23      }
24      else {
25          if (accel*TICK <= (result[0]-1 - v) && result[0]-1 >= 0) {
26              result[0]--;
27          }
28      }
29  }
30      return result;
31
32  }
```

In the contract we establish for the speed limit control method (See List. 5.3.2), KeY again only verified the method after we added explicit numbers for MAXBREAK, MAXACCEL and the TICK, but these can be replaced easily in the precondition and KeY still completes. Again we also added the complete invariant as well as safety condition from the hybrid program, that would always apply before the method was enacted as it is part of the entire system modelled by the hybrid program.

```
1 /*@ public normal_behavior
2     @ requires MAXBREAK == -10 && MAXACCEL == 10 && TICK == 1 && ((!(y
          >= slPos)) || v <= sl) && v >= 0 && sl >= 0 && (v <= sl ||
          slPos * 2 * MAXBREAK >= y*2*MAXBREAK + (v*v - sl*sl)) && ((!(sl
           < v)) || slPos * (-2) * MAXBREAK >= y*(-2)*MAXBREAK + (v*v -
          sl*sl) + (MAXACCEL + MAXBREAK) * (MAXACCEL * TICK*TICK + 2*TICK
           * v)) && ((!(sl >= v)) || accel*TICK <= (sl - v)) && (\exists
          int x;x>=0; (slPos+x) * (-2) * MAXBREAK >= y*(-2)*MAXBREAK + (v
          *v - sl*sl) + (MAXACCEL + (-1)*MAXBREAK) * (MAXACCEL * TICK*
          TICK + 2*TICK * v));
3     @ ensures \result[0] >= 0 && ((!(\result[0] < v)) || \result[1] *
          (-2) * MAXBREAK >= y*(-2)*MAXBREAK + (v*v - \result[0]*result
          [0]) + (MAXACCEL + (-1)*MAXBREAK) * (MAXACCEL * TICK*TICK + 2*
          TICK * v)) && ((!(\result[0] >= v)) || accel*TICK <= (\result
          [0] - v));
```

```
4 */
```

5.4. Glue between Java and Hybrid Program

The glue for this CPS proved interesting, as for two hook postconditions we also had to devise two proof steps. Aside from this, the complexity of the involved proof steps meant, we had to split up each proof into smaller steps as to be able to use KeYmaera's automatic verificaton mechanism. For every glue part we again quantify all the variables as $\forall\, \mathbb{R}$ (See Eq. 5.3). We also did not use the taclet we wrote for the watertank function, rather putting in the abstraction of the flooring function into our preconditions explicitly to further reduce computational complexity.

$$\begin{aligned}&\forall\, \mathbb{R}\, x_1.\forall\, \mathbb{R}\, v_1.\forall\, \mathbb{R}\, a_1.\forall\, \mathbb{R}\, v_{sl}.\forall\, \mathbb{R}\, x_{sl}.\forall\, \mathbb{R}\, B.\forall\, \mathbb{R}\, A.\forall\, \mathbb{R}\, ep.\\ &\forall\, \mathbb{R}\, x_j.\forall\, \mathbb{R}\, v_j.\forall\, \mathbb{R}\, a_j.\forall\, \mathbb{R}\, sl.\forall\, \mathbb{R}\, slPos.\forall\, \mathbb{R}\, MAXBREAK.\forall\, \mathbb{R}\, MAXACCEL.\forall\, \mathbb{R}\, TICK.\end{aligned} \tag{5.3}$$

5.4.1. Glue for the Car Control Program

For each proof part, we still follow the general glue setup (See Eq. 3.3). We create a seperate glue proof file for each part of each glue proof, as to reduce complexity for KeYmaera. For example the first part of the first glue proof (for the car control system) in the following Equation only has two glue conditions, one java postcondition and one hybrid world postcondition.

$$\begin{aligned}&(a_j - 1 < a_1 \land a_1 \leq a_j \land (a_j \geq 0 \implies a_1 \geq 0) \land (a_j < 0 \implies a_1 < 0) \land\\ &MAXBREAK - 1 < -B \land\ -B \leq MAXBREAK \land (MAXBREAK \geq 0 \implies -B \geq 0)\\ &\land (MAXBREAK < 0 \implies -B < 0) \implies\\ &((MAXBREAK \leq (a_j - 1)) \implies (-B \leq a_1)))\end{aligned}$$

The second component of the first glue part, then proves the next postcondition:

$$\begin{aligned}&(a_j - 1 < a_1 \land a_1 \leq a_j \land (a_j \geq 0 \implies a_1 \geq 0) \land (a_j < 0 \implies a_1 < 0) \land\\ &MAXACCEL - 1 < A \land A \leq MAXACCEL \land (MAXACCEL \geq 0 \implies A \geq 0)\\ &\land (MAXACCEL \leq 0 \implies A \leq 0) \implies\\ &((a_j \leq (MAXACCEL - 1)) \implies (a_1 \leq A)))\end{aligned}$$

For the third postcondition figuring out the necessary strengthening of the java postcondition proved difficult, as the computation of counterexamples by mathematica took long due to the amount of equations and conditions in the glue making finding the correct postcondition difficult. We also had to assign an explicit number to TICK/ep, as to make computation of the proof easier. We ultimately found te following glue:

$$
\begin{array}{c}
(x_1 - 1 < x_j \land x_j \leq x_1 \land \\
v_1 - 1 < v_j \land v_j \leq v_1 \land \\
a_1 - 1 < a_j \land a_j \leq a_1 \land \\
v_{sl} - 1 < sl \land sl \leq v_{sl} \land \\
x_{sl} - 1 < slPos \land slPos \leq x_{sl} \land \\
TICK = ep \land ep = 2 \land \\
(x_1 \geq x_{sl} \implies v_1 \leq v_{sl}) \land (x_j \geq slPos \implies v_j \leq sl) \implies \\
((x_j \geq slPos - 1 \implies (a_j \leq ((sl - v_j)/TICK) - 2)) \implies \\
(x_1 \geq x_{sl} \implies (a_1 \leq (v_{sl} - v_1)/ep))))
\end{array}
$$

5.4.2. Glue for the Speed Limit Control Program

The second proof part then proves the postcondition for the speed control program. Again we split it up into parts for each postcondition part to reduce computational complexity using the quantification in Eq. 5.3. We start with:

$$
\begin{array}{l}
(sl - 1 < v_{sl} \land sl \leq v_{sl} \land (sl \geq 0 \implies v_{sl} \geq 0) \land (sl < 0 \implies v_{sl} < 0)) \implies \\
(sl \geq 0) \implies (v_{sl} \geq 0)))
\end{array}
$$

The next two proof components are very similar to the ones in the first glue proof for the car control program, as the postcondition are almost identically. First we prove:

$$
\begin{array}{c}
(x_1 - 1 < x_j \land x_j \leq x_1 \land \\
v_1 - 1 < v_j \land v_j \leq v_1 \land \\
a_1 - 1 < a_j \land a_j \leq a_1 \land \\
v_{sl} - 1 < sl \land sl \leq v_{sl} \land \\
x_{sl} - 1 < slPos \land slPos \leq x_{sl} \land \\
TICK = ep \land ep = 2 \land \\
(x_1 \geq x_{sl} \implies v_1 \leq v_{sl}) \land (x_j \geq slPos \implies v_j \leq sl) \implies \\
((sl \geq v_j - 1 \implies (a_j \leq ((sl - v_j)/TICK) - 2)) \implies \\
(v_{sl} \geq v_1 \implies (a_1 \leq (v_{sl} - v_1)/ep))))
\end{array}
$$

All glue components are verified automatically by KeYmaera seperately. If put together KeYmaera does not manage the computational complexity of the glue proof, showcasing the non-triviality of the case study.

5.5. Validation of Results

After the glue we again had to make our java postcondition stronger as to be able to prove our glue. This was relatively easy due to the relative simplicity of our java control programs. The version of our java control programs includes the correct (stronger) postconditions.

6. Conclusion

In this thesis we have presented, discussed and applied our approach to refining a hybrid model of a CPS, like a hybrid program, into a concrete implementation of the control part of the system. Overall our approach appears sound during application and a viable way of gaining a concrete implementation from an existing hybrid model. Our introduced third proof component of the glue works in proving the entire CPS's correctness according to the safety property.

Finding the glue as a relation between the discrete and analogue world proved difficult during application and non-trivial even in seemingly easy systems as our first example, the watertank. As rounding or flooring functions will most likely always be a part of the glue relation due to the discrete world's , the glue proof will often have to be used to further refine the implementation. As it did for us, as the original hook postconditions used as a form of specification for the implementation will often not include the rounding/flooring errors.

Overall our work improves upon existing notions of analyzing and verifying hybrid systems and offers a non-trivial way of proving correctness of hybrid model/implementation packages.

6.1. Future Work

As mentioned in the corresponding implementation sections, our implementations for both concrete examples err on the side of caution and do not provide the same functionality a real-world example would. More case studies could be done as to show the soundness of our approach further. Aside from this, a more formalized process outline for our ownly roughly outlined approach would improve the applicability. In the future an automatic generation of the glue proof from an existing hybrid model and implementation is also thinkable.

Bibliography

[ABH+07] Wolfgang Ahrendt, Bernhard Beckert, Reiner Hähnle, Philipp Rümmer, and Peter H. Schmitt. Verifying object-oriented programs with KeY: A tutorial. In *5th International Symposium on Formal Methods for Components and Objects, Amsterdam, The Netherlands*, volume 4709 of *LNCS*, pages 70–101. Springer, 2007.

[AD94] Rajeev Alur and David L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126:183–235, 1994.

[BHS07] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach.* LNCS 4334. Springer-Verlag, 2007.

[CFH+03] Edmund Clarke, Ansgar Fehnker, Zhi Han, Bruce Krogh, Olaf Stursberg, and Michael Theobald. Verification of Hybrid Systems Based on Counterexample-Guided Abstraction Refinement. In Hubert Garavel and John Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *Lecture Notes in Computer Science*, pages 192–207. Springer Berlin Heidelberg, 2003.

[CW98] Ana Cavalcanti and Jim Woodcock. ZRC – A Refinement Calculus for Z. *Formal Aspects of Computing*, 10(3):267–289, 1998.

[DB14] John Derrick and Eerke A. Boiten. *Refinement in Z and Object-Z.* Springer London, 2014.

[GZ14] Bin Gu and Liang Zou. A refinement calculus for hybrid systems. In *Engineering of Complex Computer Systems (ICECCS), 2014 19th International Conference on*, pages 176–185, Aug 2014.

[Hen00] Thomas A. Henzinger. The theory of hybrid automata. In M.Kemal Inan and RobertP. Kurshan, editors, *Verification of Digital and Hybrid Systems*, volume 170 of *NATO ASI Series*, pages 265–292. Springer Berlin Heidelberg, 2000.

[MLP12] Stefan Mitsch, Sarah M. Loos, and André Platzer. Towards formal verification of freeway traffic control. In Chenyang Lu, editor, *ICCPS*, pages 171–180. IEEE, 2012.

[MV92] Carroll Morgan and Trevor Vickers. *On the Refinement Calculus.* Springer London, 1992.

[Pla10] André Platzer. *Logical Analysis of Hybrid Systems.* Springer, Pittsburgh, 2010.

[Pla15a] André Platzer. Guide for Keymaera Hybrid Systems Verification Tool. http://symbolaris.com/info/KeYmaera-guide.html, 2015.

[Pla15b] André Platzer. KeYmaera: A Hybrid Theorem Prover for Hybrid Systems. http://symbolaris.com/info/KeYmaera.html, 2015.

[RT13] Matthias Rungger and Paulo Tabuada. Abstracting and refining robustness for cyber-physical systems. *CoRR*, abs/1310.5199, 2013.

[RyS96] Jean-François Raskin and Pierre yves Schobbens. State Clock Logic: a Decidable Real-Time Logic. In *HART'97, LNCS 1201*, pages 33–47. Springer-Verlag, 1996.

[STW14] Steve Schneider, Helen Treharne, and Heike Wehrheim. The behavioural semantics of Event-B refinement. *Formal Aspects of Computing*, 26(2):251–280, 2014.

[Wol] Wolfram Research, Inc. Mathematica. https://www.wolfram.com.

A. Appendix

In this Appendix we present images, complete source code/hybrid program code listings as well as a Glossary at the end.

A.1. Images

In the following section different images that have been excluded from the main thesis are presented.

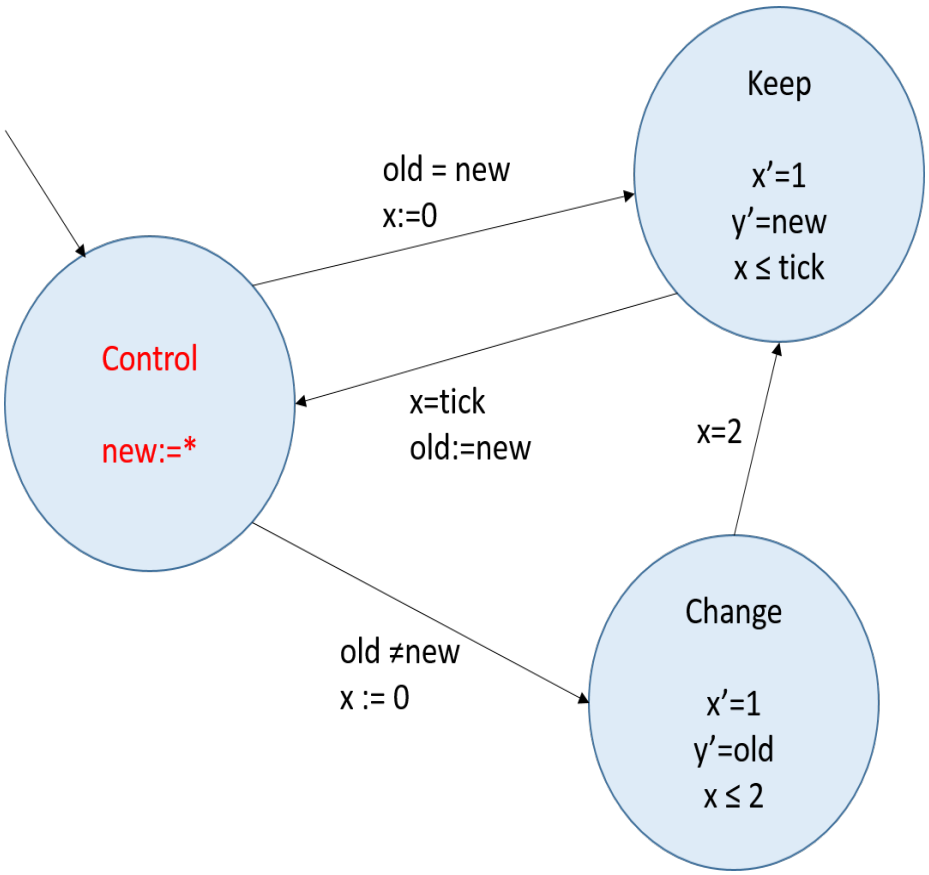

Figure A.1.: Watertank Hybrid Automata with Non-Deterministic Control Program Abstraction marked.

A.2. Listings

In the following section we present the full java source code for our control program classes, as well as the rule definition we used for KeYmaera. First is the rule definition for KeYmaera, that abstracts the application of the floor function to a inequality of the form $\text{floor}(x) = y \equiv \exists c : x - 1 < c \land c \leq x \land (x \geq 0 \implies c \geq 0) \land (x < 0 \implies c < 0) \longrightarrow$ whereby $\longrightarrow$ refers to the big implication arrow in the verification of a hybrid program, meaning we get the abstraction of $f(x)$ on the left side of the implication.

```
\rules {
   fdef {
      \schemaVar \term R x;
      \schemaVar \skolemTerm R c;
         \find(f(x))
      \sameUpdateLevel
      \varcond ( \new(c, \dependingOn(x)) )
      \replacewith(c)
      \add(x-1 < c & c <= x & (x >= 0 -> c >= 0) & (x < 0 -> c < 0)
         ==> )
      \heuristics(simplify)
   };
}
```

Next, we provide the full java source code for the Watertank control program class. This means, not only the control method itself is provided, but all the needed class definitions as well.

```
package watertankSplit;

/**
 * @author Daniel Draper
 * @version 1.0
 * This class is the Watertank's control system.
 */
public class Controller {
   private int tick;

   public Controller() {
      tick = 30;
   }

   /**
    * Returns the actual control value for the valve of the watertank.
    * @param y the current water level
    * @param old the old valve setting
    * @return the new valve setting
    */
   /*@ public normal_behavior
     @ requires tick == 31 && y >= 10 && y <= 120 && y + 2 * old <= 120
         && y + 2 * old >= 10 && (old == 10 || old == -20);
     @ ensures \result * (tick / 10) + y + 2 * old  >= 12 & \result * (
         tick / 10) + y + 2 * old <= 116 && (\result == 10 || \result ==
          -20);
```

```
24      @*/
25    public int getControlValue (int y, int old) {
26       //Waterlevel in two time units
27       int inTwo = y + 2 * old;
28       //If we are raising level, keep raising if possible without
             hitting max_level before next tick
29       if (old == 10) {
30          if (inTwo + tick * 1 <= 116) {
31             return 10;
32          }
33          else {
34                return -20;
35          }
36       }
37       //ELSE if we are currently lowering level, keep lowering if we can
              lower further without hitting min_level b4 next tick
38       else {
39          if (old == -20) {
40             if (inTwo - tick * 2 >= 12) {
41                return -20;
42             }
43             else {
44                   return 10;
45
46             }
47
48          }
49       }
50       //Only returned if old != 10 && old != -20, unreachable.
51       return 0;
52    }
53 }
```

Next, you can find the full source code for the Traffic Control Car Control Program.

```
1  package trafficControl;
2
3  /**
4   * @author Daniel Draper
5   * @version 1.0
6   * This class models the Car Controller of the Traffic Control CPS.
7   */
8  public class CarController {
9     private int TICK;
10    private int MAXBREAK;
11    private int MAXACCEL;
12
13    public CarController(int TICK, int MAXBREAK, int MAXACCEL) {
14       this.TICK = TICK;
15       this.MAXBREAK = MAXBREAK;
16       this.MAXACCEL = MAXACCEL;
17    }
18    /**
19     * @param slPos Current Speed Limit Position
```

```
20      * @param sl current Speed Limit
21      * @param y current Car Position
22      * @param v current Car Velocity
23      * @param accel current Car acceleration
24      * @return the new acceleration control Value
25      * The actual control Method.
26      */
27     /*@ public normal_behavior
28       @ requires MAXBREAK <= accel && accel <= MAXACCEL && ((!(y >= slPos
             )) || v <= sl) && v >= 0 && sl >= 0 && TICK == 1 && MAXBREAK ==
              -10 && MAXACCEL == 10 && 2 * MAXBREAK * (slPos - y) >= v*v -
             sl*sl && (v <= sl || slPos * 2 * MAXBREAK >= y*2*MAXBREAK + (v*
             v - sl*sl));
29       @ ensures MAXBREAK <= \result && \result <= MAXACCEL && ((!(y >=
             slPos)) || (\result*TICK <= (sl - v))) && ((!(y < slPos)) ||
             (2*slPos*MAXBREAK*(-1) >= y*2*(-1)*MAXBREAK + (v*v - sl*sl)  +
             (\result+(-1)*MAXBREAK) * (\result * TICK*TICK + 2*TICK * v)));
30      @*/
31     public int control(int y, int v, int accel, int sl, int slPos) {
32        if (y >= slPos) {
33           if ((accel+1)*TICK <= (sl-v)) {
34              if (accel+1 <= MAXACCEL) {
35                 return accel+1;
36              }
37
38           }
39           else {
40              if((accel-1)*TICK <= (sl-v)) {
41                 if (accel-1 >= MAXBREAK) {
42                    return accel-1;
43                 }
44           }
45           }
46        }
47        else {
48           if (2*slPos*MAXBREAK*(-1) >= y*2*(-1)*MAXBREAK + (v*v - sl*sl)
                + ((accel+1)+(-1)*MAXBREAK) * ((accel+1) * TICK*TICK + 2*
              TICK * v)) {
49              if (accel+1 <= MAXACCEL) {
50                 return accel+1;
51              }
52           }
53           else {
54              if (2*slPos*MAXBREAK*(-1) >= y*2*(-1)*MAXBREAK + (v*v - sl*
                   sl)  + ((accel-1)+(-1)*MAXBREAK) * ((accel-1) * TICK*
                   TICK + 2*TICK * v)) {
55                 if (accel-1 >= MAXBREAK) {
56                    return accel-1;
57                 }
58              }
59           }
60        }
61           return MAXBREAK;
62
```

```
63    }
```

Next, you can find the full source code for the Speed Limit Control Class.

```
1
2  package trafficControl;
3
4
5  /**
6   * @author Daniel Draper
7   * @version 1.0
8   * This class models the Speed Limit Controller used by the Traffic
        Control CPS.
9   */
10 public class SpeedLimitController {
11
12    private int TICK;
13    private int MAXBREAK;
14    private int MAXACCEL;
15    /**
16     * @param TICK the Tick duration
17     * @param MAXBREAK the maximum breaking power (<0)
18     * @param MAXACCEL the maximum acceleration (>=0)
19     */
20    public SpeedLimitController(int TICK, int MAXBREAK, int MAXACCEL) {
21       this.TICK = TICK;
22       this.MAXBREAK = MAXBREAK;
23       this.MAXACCEL = MAXACCEL;
24    }
25
26    /**
27     * @param slPos the current speed limit position
28     * @param sl the current speed limit
29     * @param accel the current car's acceleration value
30     * @param v the current car's velocity
31     * @param y the current car's position
32     * @return both the new speed limit and speed limit position in an
          array.
33     * The actual control Method.
34     */
35    /*@ public normal_behavior
36      @ requires MAXBREAK == -10 && MAXACCEL == 10 && TICK == 1 && ((!(y
            >= slPos)) || v <= sl) && v >= 0 && sl >= 0 && (v <= sl ||
            slPos * 2 * MAXBREAK >= y*2*MAXBREAK + (v*v - sl*sl)) && ((!(sl
             < v)) || slPos * (-2) * MAXBREAK >= y*(-2)*MAXBREAK + (v*v -
            sl*sl) + (MAXACCEL + MAXBREAK) * (MAXACCEL * TICK*TICK + 2*TICK
             * v)) && ((!(sl >= v)) || accel*TICK <= (sl - v)) && (\exists
            int x;x>=0; (slPos+x) * (-2) * MAXBREAK >= y*(-2)*MAXBREAK + (v
            *v - sl*sl) + (MAXACCEL + (-1)*MAXBREAK) * (MAXACCEL * TICK*
            TICK + 2*TICK * v));
37      @ ensures \result[0] >= 0 && ((!(\result[0] < v)) || \result[1] *
            (-2) * MAXBREAK >= y*(-2)*MAXBREAK + (v*v - \result[0]*result
            [0]) + (MAXACCEL + (-1)*MAXBREAK) * (MAXACCEL * TICK*TICK + 2*
```

```
        TICK * v)) && ((!(\result[0] >= v)) || accel*TICK <= (\result
        [0] - v));
38    */
39    public int[] control(int y, int v, int accel, int sl, int slPos) {
40       int[] result = new int[2];
41       result[0] = sl;
42       result[1] = slPos;
43       if (sl<25) {
44          if (sl+1 < v) {
45             result[0]++;
46             result[1] = y + (v*v - result[0]) + (MAXACCEL + 1) * (
                  MAXACCEL * TICK*TICK + 2 * TICK * v);
47          }
48          else {
49             if (accel*TICK <= result[0]+1 - v) {
50                result[0]++;
51             }
52          }
53       }
54       else {
55       if (result[0]-1 < v) {
56          if (result[0]-1 >= 0) {
57          result[0]--;
58
59          result[1] = y + (v*v - result[0]) + (MAXACCEL + 1) * (MAXACCEL
                * TICK*TICK + 2 * TICK * v);
60          }
61       }
62       else {
63          if (accel*TICK <= (result[0]-1 - v) && result[0]-1 >= 0) {
64             result[0]--;
65          }
66       }
67    }
68       return result;
69
70    }
71 }
```

A.3. Watertank simulator

To be able to execute the watertank control programs and see a result of the computations done on screen (in a form of console output) we had to write a simulator of the differential evolution of the watertank values. The watertank simulator class can be found below:

```
1 package watertankSplit;
2
3 /**
4  * @author Daniel Draper
5  * @version 1.0
6  * This class simulates a watertank as modelled in the hybrid model in
     the KeYmaera Guide.
```

```
7  */
8
9  public class Simulator
10 {
11    private int TICK = 30;
12    private Controller contr;
13    private int y;
14    private int x;
15    private int state;
16    private int oldDif;
17    private int newDif;
18
19    /**
20     * Constructs the simulator with correct starting values according to
          hybrid model.
21     */
22    public Simulator() {
23       contr = new Controller();
24       state = 1;
25       y = 10;
26       x = TICK;
27       oldDif = 10;
28       newDif = 10;
29    }
30
31    /**
32     * One step in the differential evolution of the watertank.
33     */
34    private void step() {
35       switch (state) {
36       case 0:
37          if (x == 20) {
38             state = 1;
39          }
40          else {
41             y+=oldDif;
42             x+=10;
43          }
44          break;
45       case 1:
46          if (x == TICK) {
47             oldDif = newDif;
48             System.out.println("Controlled!");
49             newDif = contr.getControlValue(y, oldDif);
50             if (newDif == oldDif) {
51                state = 1;
52                x = 0;
53             }
54             else {
55                state = 0;
56                x = 0;
57             }
58          }
59          else {
```

```
60              y+=newDif;
61              x+=10;
62          }
63          break;
64      }
65      }
66
67      /**
68       * Continuous execution of each timestep and printing the state with
             an included delay.
69       */
70      public void run() {
71          do {
72              step();
73              printState();
74                  synchronized(this) {
75                      try {
76                          this.wait(500);
77                      } catch (InterruptedException e) {
78                          e.printStackTrace();
79                      }
80                      finally {}
81                  }
82          } while(true);
83      }
84      /**
85       * Prints the current state of the Watertank.
86       */
87      private void printState() {
88          System.out.println("Time: " + x/10 + " newDif: " + newDif + "
                 Fillstatus:");
89          for (int i = 0; i < y/10; i++) {
90          System.out.print("X ");
91          }
92          for (int i = 0; i < 12 - y/10; i++) {
93              System.out.print("_ ");
94          }
95          System.out.println();
96
97      }
98
99      /**
100      * Main function creating and executing a new Simulator.
101      * @param args commandline arguments
102      */
103     public static void main(String args[]) {
104         Simulator s = new Simulator();
105         s.run();
106     }
107
108 }
```

A.4. Traffic Control Simulator

As we did for the Watertank CPS, we also wrote a Simulator for the Traffic Control CPS as to be able to execute the entire program and see the evolution of the car. As the amount of distance between the next speed limit and current car's position can vary greatly, we chose a different way to present the current status of the system than for the watertank, just listing the current status in text form and checking if safety is still guaranteed. It consists of an infinite loop always continuing with one step in the differential evolution of the system, printing the state and an added delay as to make console output readable.

```
1
2  package trafficControl;
3
4  /**
5   * @author Daniel Draper
6   * @version 1.0
7   * This class simulates the environment in the Traffic Control CPS.
8   */
9  public class Simulator {
10
11     private int xDif;
12     private int x;
13     private int y;
14     private int v;
15     private int accel;
16     private int sl;
17     private int slPos;
18     private CarController carContr;
19     private SpeedLimitController limContr;
20     private static int TICK = 1;
21     private static int MAXACCEL = 10;
22     private static int MAXBREAK = -10;
23
24     /**
25      * Creates a new Traffic Control Simulator.
26      */
27     public Simulator() {
28        xDif = 1;
29        v = 10;
30        x = 0;
31        y = 0;
32        slPos = 10;
33        sl = 10;
34        carContr = new CarController(TICK, MAXBREAK, MAXACCEL);
35        limContr = new SpeedLimitController(TICK,MAXBREAK,MAXACCEL);
36     }
37
38     /**
39      * Starts the entire system, running in an infinite loop.
40      */
41     public void run() {
42        do {
43           printState();
```

```
44          if (x \% TICK == 0) {
45              accel = carContr.control(y,v,accel,sl,slPos);
46                  int[] res = limContr.control(y,v,accel,sl,slPos);
47                  sl = res[0];
48                  slPos = res[1];
49          }
50          int v0 = v;
51          x += xDif;
52          v += accel;
53          y += (v0 + 0.5 * accel);
54              synchronized(this) {
55          try {
56              this.wait(500);
57          } catch (InterruptedException e) {
58              // TODO Auto-generated catch block
59              e.printStackTrace();
60          }
61          finally {}
62      }
63      } while(true);
64  }
65  /**
66   * Prints the current state of the system.
67   */
68  private void printState() {
69      System.out.println("We have: Car at Pos. " + y + " with velocity "
            + v + " and acceleration " + accel);
70      System.out.println("Speed limit " + sl + " at Pos. " + slPos);
71      System.out.println("If car breaks at max_Break_Speed " + MAXBREAK
            + " at next tick, car has speed " + (v + accel) + " and
            afterwards speed " + (v+accel+MAXBREAK));
72      if (y >= slPos) {
73          if (accel*TICK <= (sl- v)) {
74              System.out.println("Safety kept!");
75              return;
76          }
77      }
78      else {
79          if (2*slPos*MAXBREAK*(-1) >= y*2*(-1)*MAXBREAK + (v*v - sl*sl)
                + (accel+(-1)*MAXBREAK) * (accel * TICK*TICK + 2*TICK * v)
                ); {
80              System.out.println("Safety kept!");
81              return;
82          }
83      }
84      System.err.println("Safety condition not kept!");
85  }
86
87  /**
88   * @param args Commandline Arguments.
89   * Main function for execution.
90   */
91  public static void main(String[] args) {
92      Simulator s = new Simulator();
```

```
93          s.run();
94      }
95  }
```

A.5. Complete Hybrid Programs in correct ASCII notation

In this section we present the complete hybrid programs from this thesis. As we mentioned in Ch. 2, hybrid programs have to be typeset in ASCII for KeYmaera to be able to read it, which is why these hybrid programs look different from the ones we present in the previous chapters. The first two hybrid programs are the original watertank hybrid program and our remodelled version that includes the hook, a suitable postcondition as well as the tick.

```
\functions {
}

\programVariables {
    R y, x, st;
}

\problem {
    /* initialization */
    \[ x:=0; y:=1; st:=0 \] ( (st = 0) /*initial state characterization */
        ->
    \[ /* system dynamics */
        ( /* repeat the discrete/continuous transitions */
        (?(st=0);
            (?(y = 10); x:=0; st:=1)
            ++ (?(y < 10 | y > 10); {x'=1,y'=1, y<=10})
            )
        ++
        (?(st=1);
            (?(x=2); st:=2)
            ++ (?(x < 2 | x > 2); {x'=1,y'=1, x <=2})
        )
        ++ (?(st=2);
            (?(y=5); x:=0; st:=3)
            ++ (?(y>5 | y < 5); {x'=1, y'=-2, y >=5})
        )
        ++ (?(st=3);
            (?(x=2); st:=0)
            ++ (?(x>2 | x < 2); {x'=1,y'=-2, x <= 2})
        )
        )*@invariant(y >=1 & y <=12 & (st=3 -> (y >= 5 - 2*x)) & (st=1->(y<=10+x)))
    \]  (y >= 1 & y <= 12)) /*safety postcondition */
}
```

```
\functions {
    /*tick is constant*/
    R tick;
}

\programVariables {
    /* variables in use */
    R y; R x; R new; R valve;
}


\problem {
    /*requirement from our model*/
    tick > 2  ->
    \[
            /*initialization*/
            x := tick; y := 1; valve := 1;
            /*hook: new:= * */
            ((?(x = tick ); new := *; x:= 0);
            /*safety hook postcondition*/
            (?(y + 2 * valve + tick * new >= 1 & y + 2 * valve + tick * new <= 12 & (new = 1 |
            new = -2));

            ((?(new != valve); {x' = 1, y' = valve & x <= 2}); if (x=2) then valve := new; {x'
            = 1, y' = valve & x <= tick} fi)

            ++ (?(!(new != valve)); {x' = 1, y' = valve & x <= tick})

            ))*@invariant(y >=1 & y <=12 & (valve = -2 | valve = 1) & (x = tick -> (y + 2*
            valve >= 1 & y + 2* valve  <= 12)))
    \] /*safety condition*/(y >= 1 & y <= 12)
}
```

Next up is the glue proof for the watertank cps typeset in ASCII.

```
\functions {
        R f(R);

}

\programVariables {

R tick; R y; R x; R new; R valve; R oldj; R result; R tickj; R yj;
}


\rules {
  fdef {
    \schemaVar \term R x;
    \schemaVar \skolemTerm R c;
    \find(f(x))
    \sameUpdateLevel
    \varcond ( \new(c, \dependingOn(x)) )
    \replacewith(c)
    \add(x-1 < c & c <= x & (x >= 0 -> c >= 0) & (x < 0 -> c < 0) ==> )
    \heuristics(simplify)
    };

}

\problem {

(\forall R y . \forall R yj . \forall R valve .\forall R tick .\forall R tickj .\forall R new
.\forall R result .
 ((yj = f(10 * y)  & oldj =  f(10 * valve) & tickj = f(10 * tick) & result = 10 * new) & y >= 1
 & y <= 12 & (valve = 1 | valve = -2) & tick  > 2 ->

((result * tickj/10 + yj + 2 * oldj <= 116 & result * tickj/10 + yj + 2 * oldj >= 12 & (result
= 10 | result = -20)) ->
(y + 2 * valve + tick * new >= 1 & y + 2 * valve + tick * new <= 12 & (new = 1 | new = -2)))))

}
```

The next two hybrid programs are the original Traffic Control hybrid program, as well as it's remodelled version that includes the hooks (as in this case we have two control programs), the postconditions and a tick.

```
\programVariables {
  R x1, v1, a1, t;  /* car 1 */
  R vsl, xsl;       /* traffic center */
  R B, A, ep;       /* system parameters */
}
/**
 * One lane, one car, one traffic center. Traffic center may issue speed limits at any time.
 Car needs up to ep time units to react (includes communication).
 * Car can brake and accelerate.
 * Checks if car complies with the speed limit after point xsl.
 */
\problem {
    (   v1 >= 0
        & vsl >= 0
        & x1 <= xsl
        & 2 * B * (xsl - x1) >= v1^2 - vsl^2
        & A >= 0
        & B > 0
        & ep > 0
      -> \[ (
                /* control car */
                (a1 :=
                -B)
                /* braking is always allowed */
                ++ (?xsl >= x1 + (v1^2 - vsl^2) / (2 * B) + (A / B + 1) * (A / 2 * ep^2 + ep *
                v1);     /* outside the speed limit do whatever you want, as long as you can
                still brake to meet the speed limit */
                    a1 := *;  ?-B <= a1 & a1 <=
                    A)
                ++ (?x1 >= xsl; a1 := *; ?-B <= a1 & a1 <= A & a1 <= (v1 - vsl) /
                ep);                  /* comply with the speed limit by not accelerating too
                much */

                /* traffic center, keep previous or set a new speed limit */
                (xsl := xsl; vsl := vsl)
                ++ (xsl := *; vsl := *; ?vsl >= 0 & xsl >= x1 + (v1^2 - vsl^2) / (2 * B) + (A /
                B + 1) * (A / 2 * ep^2 + ep * v1)); /* if we set a speed limit, the car must be
                able to comply with it, no matter how hard it currently accelerates */

                t := 0;
                /* dynamics */
                {x1' = v1, v1' = a1, t' = 1, v1 >= 0, t <= ep}
            )*
            @invariant(v1 >= 0 & vsl >= 0 & (v1 <= vsl | xsl >= x1 + (v1^2 - vsl^2) / (2 *
            B)))
        \] (x1 >= xsl -> v1 <= vsl)
    )
}
```

```
\programVariables {
  R x1, v1, a1, t;  /* car 1 */
  R vsl, xsl;       /* traffic center */
  R B, A, ep;       /* system parameters */
}
/**
 * One lane, one car, one traffic center. Traffic center may issue speed limits at any time.
 Car needs up to ep time units to react (includes communication).
 * Car can brake and accelerate.
 * Checks if car complies with the speed limit after point xsl.
 */
\problem {
    (   v1 >= 0
        & vsl >= 0
        & x1 <= xsl
        & 2 * B * (xsl - x1) >= v1^2 - vsl^2
        & A >= 0
        & B > 0
        & ep > 0
      -> \[ (
                /* control car */
                ?(t=ep);
                (a1 := *);
                ?(-B < a1 & a1 < A & (x1 >= xsl -> (a1 <= (vsl - v1) / ep)) & (x1 < xsl -> (xsl
                >= x1 + (v1^2 - vsl^2) / (2 * B) + (a1 / B + 1) * (a1 / 2 * ep^2 + ep * v1))));

                xsl := *; vsl := *;
                ?(vsl >= 0 & xsl >= x1 + (v1^2 - vsl^2) / (2 * B) + (A / B + 1) * (A / 2 * ep^2
                + ep * v1));
                t := 0;
                /* dynamics */
                {x1' = v1, v1' = a1, t' = 1, v1 >= 0, t <= ep}
            )*
            @invariant(v1 >= 0 & vsl >= 0 & (v1 <= vsl | xsl >= x1 + (v1^2 - vsl^2) / (2 *
            B)))
        \] (x1 >= xsl -> v1 <= vsl)
    )
}
```

What now follows is the glue proof for the traffic control cps typeset in ASCII.

Glossary

Cyber-Physical System[CPS] is a system describing motions or evolutions in which a physical aspect is being controlled by a computer/computer program. In this thesis equivalent to the notion of Hybrid Systems.

Differntial Dynamic Logic[DDL] is the logic in which we express the safety properties for our CPS, and it also includes a syntax to express differential equations.

Dynamic Logic is the logic we use to express the safety property for our CPS.

Glue is the relation between the values in the world of reals and the discrete world. For us, refers to a way of gaining the corresponding value in the other world from a given value.

Hook is the concrete instruction at which the control program is executed when describing a hybrid system as a hybrid program. This is one or multiple non-deterministic assignments of values, e.g a:=*.

Hook Safety Postcondition is the condition that has to be fullfilled by the value(s) that were assigned in the hook for the safey condition of the whole program to hold true.

Hybrid Automata is a way to model hybrid systems in the form of a non-deterministic automata. Uses the same syntax as finite automata with the addition of differential equations.

Hybrid Program is a way to describe hybrid systems in the form of a program. Expressed in the syntax of regular programs with the extension of differential equations.

Hybrid System is a system in which discrete as well as continuous evolutions are present. E.g, a remote controlled car which can only be accelerated or braked, its movement is continuous and follows continuous differential equations, while the control program is discrete and can only take discrete values (e.g, Acceleration := 1; Acceleration := 2 etc.).

Java Modelling Language is the language we use to express the contracts a certain method or class has to fulfill to be considered correct.

KeY is the tool we use to verify our java control programs as our concrete implementations.

KeYmaera is the tool we use to verify both our remodelled hybrid programs as well as the glue relation.