Karlsruhe Institute of Technology

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.

PLACE, DATE

. . . .

(Daniel H. Draper)

Abstract

English abstract.

Zusammenfassung

Deutsche Zusammenfassung

Contents

Abstract i

Zusammenfassung iii

1. Introduction 1
- 1.1. Related Work . . . 2
- 1.2. Outline . . . 2

2. Preliminary Definitions 3
- 2.1. Introducing Cyber-Physical-Systems . . . 3
  - 2.1.1. Modelling CPS as Hybrid Automata . . . 3
  - 2.1.2. Modelling CPS as Hybrid Programs . . . 4
- 2.2. Dynamic Logic . . . 5
  - 2.2.1. Differential Dynamic Logic as our Verification Logic . . . 5
  - 2.2.2. JML: Verification of Java Programs using KeY . . . 6

3. Motivating Example: Refinement of CPS "Watertank" 7
  - 3.0.3. The original Hybrid Program . . . 8
- 3.1. Finding the concrete Control Value Assisgnment . . . 9
  - 3.1.1. The Remodelled Hybrid Program . . . 9
- 3.2. Finding the correct Program Safety Condition . . . 10
- 3.3. The (simple) Java Control Program . . . 11
- 3.4. Finding the glue between Java and the Hybrid Model of the system . . . 12
- 3.5. Verification based on KeYmaera . . . 13

4. Formalized approach of using refinement to gain an implementation for a CPS 15
- 4.1. Modelling CPS as Hybrid System that includes concrete Hook . . . 16
- 4.2. Finding the necessary safety condition of the control value for verification with KeYmaera . . . 16
- 4.3. Implementing control program according to safety condition as its specification and verification by KeY. . . . 16
- 4.4. Finding the correct glue between hybrid model and control program and verifying it with KeYmaera. . . . 17
- 4.5. Validating glue results against implementation . . . 17

5. Evaluation 19
- 5.1. First Section . . . 19
- 5.2. Second Section . . . 19
- 5.3. Third Section . . . 19

6. Conclusion **21**

A. Appendix **25**
A.1. Images 25
A.2. Listings 25
A.3. Watertank simulator 25
A.4. Complete Hybrid Programs in correct notation 25

List of Figures

2.1. A simple Hybrid Automata [?]. 4
2.2. A simplified train control system as a Hybrid Program [Pla10]. 5

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]. . . 7
3.2. The Watertank CPS expressed as a hybrid automata [Pla15a]. 8

A.1. Watertank Hybrid Program and - Automata with Non-Deterministic Control Program Abstraction marked. 26

List of Tables

2.1. Syntax of Hybrid Programs (HP) [Pla10]. 5

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. 3.4), 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

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: another case study: CPS “Traffic Control” (See ref. [**?**]).

2. Preliminary Definitions

In the following chapter we want to define and terms, models and logic used throughout this thesis. Our explanations are based on the premise, that first-order logic is a known concept.

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. [?].

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.

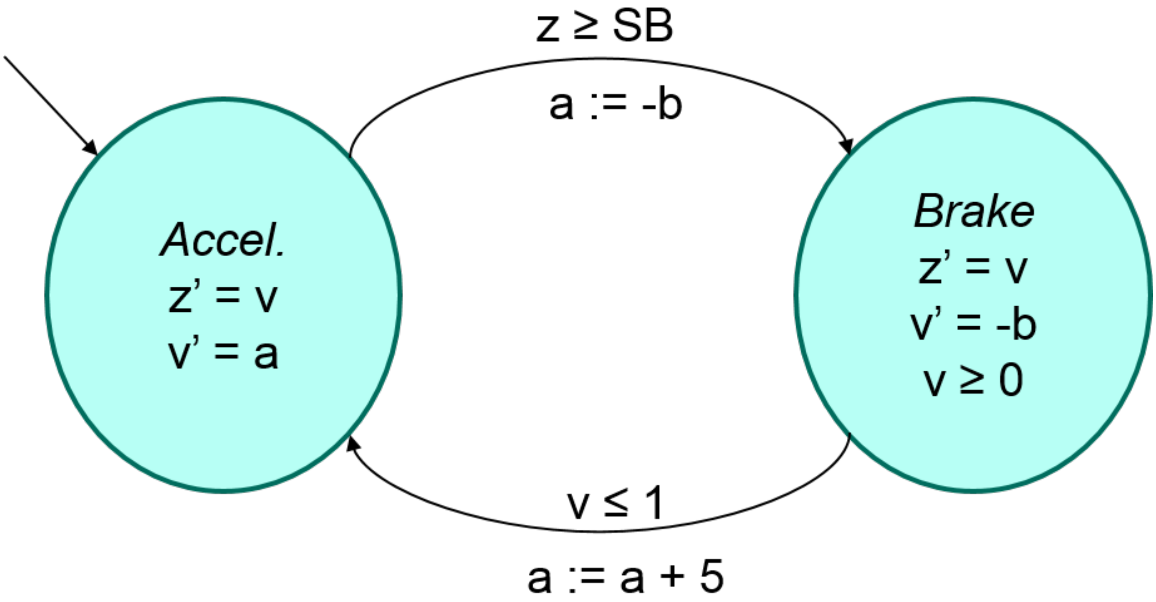


Figure 2.1.: A simple Hybrid Automata [?].

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 $x \in V$
$x_1' = \theta_1 \land \ldots$ $\ldots \land x_n' = \theta_n \land \chi$	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).

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

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* (⟨⟩), 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 as our Verification Logic

While dynamic logic in itself is not bound to only be applicable to 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. JML is then listed as a comment in

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.

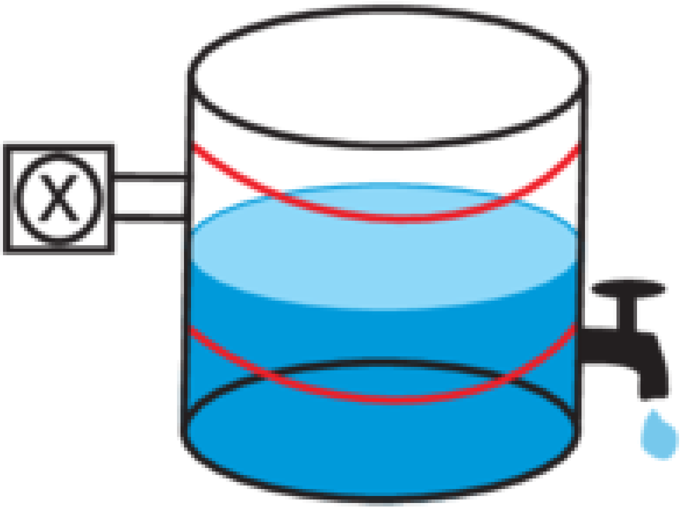


Figure 3.1.: Picture of possible Watertank configuration: Watertank that can get drained at all times and has a valve controlled by a control program [Pla15a].

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. 3.0.3). In the hybrid automaton, the system starts in the "fill" state, where water is constantly flowing into the tank with 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

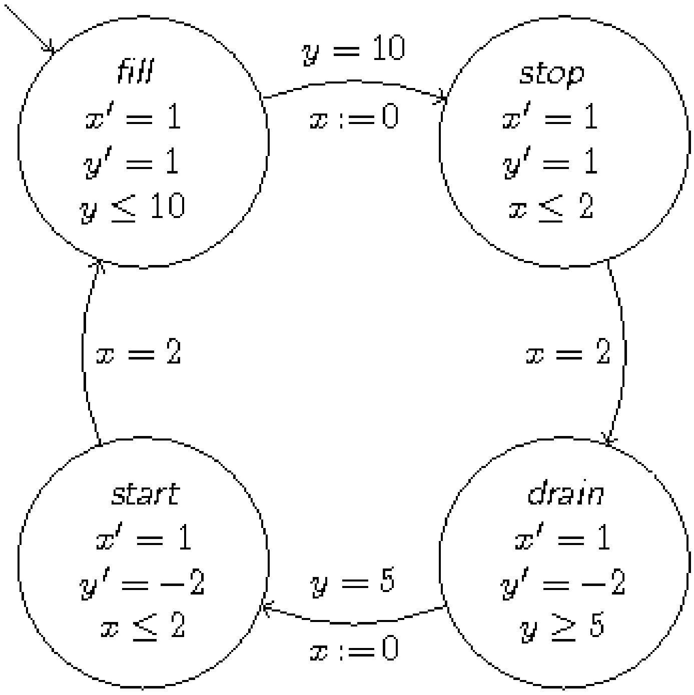

fill
$x' = 1$
$y' = 1$
$y \leq 10$
$y = 10$
$x := 0$
stop
$x' = 1$
$y' = 1$
$x \leq 2$
$x = 2$
$x = 2$
start
$x' = 1$
$y' = -2$
$x \leq 2$
$y = 5$
$x := 0$
drain
$x' = 1$
$y' = -2$
$y \geq 5$

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

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 $y = 5$.

3.0.3. The original Hybrid Program

The hybrid program (see App. A.4 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 notatio with a preceeding initialization of values and state.

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

3.1. Finding the 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. Therefore, we remodelled the system to better serve our purpose, featuring a clear ticked hook that is called upon at deterministic times (See sec 3.1.1).

3.1.1. The Remodelled Hybrid Program

The remodelled hybrid program(see App. A.4 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 $\in \mathbb{R}$.

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

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.

3.2. Finding the correct Program Safety Condition

Next we devised a postcondition for the program hook, which would then serve as the abstraction of the java control program implementation. This means, that the program would be built accordingly, so that it could be verified against this postcondition. 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).

$$\begin{aligned}
\psi \equiv (y + 2 * valve + tick * new \geq 1 \land \\
y + 2 * valve + tick * new \leq 12 \land \\
(new = 1 \lor new = -2))
\end{aligned} \tag{3.2}$$

KeYmaera verified our new hybrid program fully automatically.

3.3. 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. 3.3).

```java
public int getControlValue (int y, int old) {
        //Waterlevel in two time units
        int inTwo = y + 2 * old;
        //If we are raising level, keep raising if possible without hitting max_level before next tick
        if (old == 10) {
            if (inTwo + tick * 1 <= 116) {
                return 10;
            }
            else {
                    return -20;
            }
        }
        //ELSE if we are currently lowering level, keep lowering if we can lower further without hitting min_level b4 next tick
        else {
            if (old == -20) {
                if (inTwo - tick * 2 >= 12) {
                    return -20;
                }
                else {
                        return 10;

                }

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

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. 3.3). 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 $tick == 30;\ tick == 31; \ldots$ assignments.

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

```
  /*@ public normal_behavior //
    @ requires tick == 30 && 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); //
    @*/
```

3.4. Finding the glue between Java and the Hybrid Model of the system

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.

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

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:

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

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

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

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

$$\begin{gathered}y + 2 * valve + tick * new \geq 1 \land y + 2 * valve + tick * new \leq 12 \land \\ (new = 1 \lor new = -2)\end{gathered} \tag{3.6}$$

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. 3.3). 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.

3.5. 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. ??).

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

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 using refinement to gain an implementation for a CPS

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 CPS as Hybrid System that includes concrete 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. 3.1, 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, and model a form of tick, so as to make an implementation actually feasible.

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

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. Implementing control program according to safety condition as its specification and verification by KeY.

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. 3.4) 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. Finding the correct glue between hybrid model and control program and verifying it with KeYmaera.

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:

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

Coming up with the glue is the hardest part of the entire process, and for us 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 without lots of complex mathematical computations, making the proof hard for humans.

We used a form of logical deduction to come up with the correct quantifiers for the glue proof (so the ∀ 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 and the real-world postcondition ϕ that we came up with in step 3, and the glue that we devise which is verified by KeYmaera, we reach the goal we set out for (See Eq. 1).

4.5. Validating glue results against implementation

As we realized in our Watertank example (See Sec. 3.4), 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 according to Eq. 1).

5. Evaluation

…

5.1. First Section

…

5.2. Second Section

…

5.3. Third Section

…

6. Conclusion

…

Bibliography

[Hen00] ThomasA. 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.

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

[Pla15a] André Platzer. Guide for Keymaera Hybrid Systems Verification Tool, 2015.

[Pla15b] André Platzer. KeYmaera: A Hybrid Theorem Prover for Hybrid Systems, 2015.

[Wol] Wolfram Research, Inc. Mathematica.

A. Appendix

A.1. Images

A.2. Listings

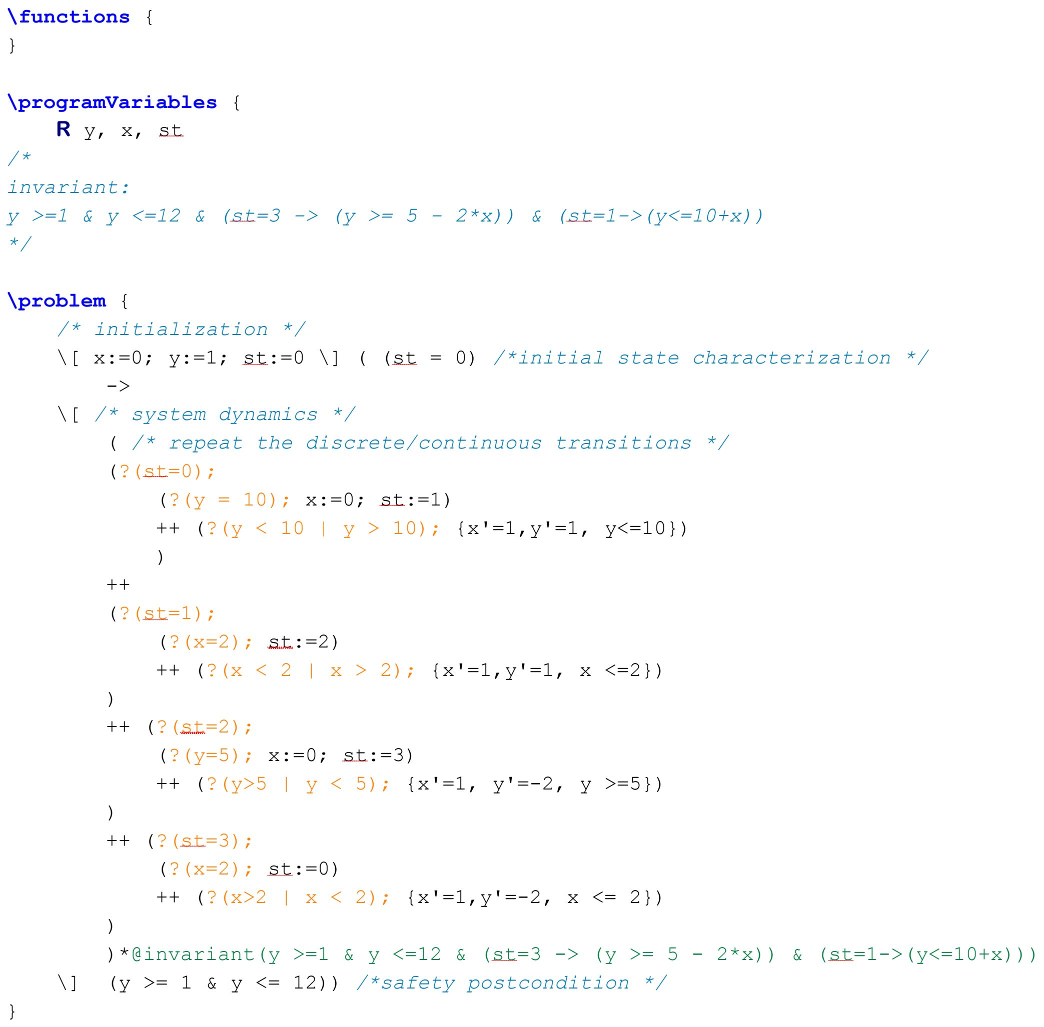
```
\rules {
    fdef {
        \schemaVar \term R x;
        \schemaVar \skolemTerm R c;
            \find(f(x))
        \sameUpdateLevel
        \varcond ( \new(c, \dependingOn(x)) )
        \replacewith(c)
        \add(x-1 < c & c <= x & (x >= 0 -> c >= 0) & (x < 0 -> c < 0)
          ==> )
     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

A.3. Watertank simulator

A.4. Complete Hybrid Programs in correct notation

Placeholder

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



```
\functions {
}

\programVariables {
    R y, x, st
/*
invariant:
y >=1 & y <=12 & (st=3 -> (y >= 5 - 2*x)) & (st=1->(y<=10+x))
*/

\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;
    \external R Floor(R); /* using mathematica floor function does not work */
}

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

/*
invariant:
y >=1 & y <=12
*/

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