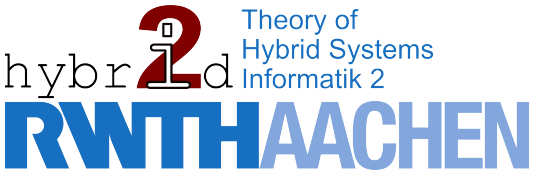

RWTH Aachen University
Rheinisch-Westfälische Technische Hochschule Aachen

Lehr- und Forschungsgebiet
Theorie hybrider Systeme
Prof. Dr. Erika Ábrahám

Master Thesis

Modular Verification for PLC Controlled Hybrid Systems

Kai Axel Driessen
Matriculation Number: 297607

- September 2014 -

Primary Referee:	Prof. Dr. Erika Ábrahám
Secondary Referee:	Prof. Dr. Thomas Noll
Supervisors:	Dipl.-Inform. Johanna Nellen
	Dr. Martin R. Neuhäußer

Declaration of Academic Integrity

I hereby declare that I have created this work completely on my own and used no other sources or tools than the ones listed, and that I have marked any citations accordingly.

Hiermit versichere ich, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

_____

Aachen, the 30th of September 2014

Acknowledgments

At this point, I would like to thank Prof. Dr. Erika Ábrahám for giving me the opportunity to write my master thesis at the Chair of Computer Science 2 and for being primary referee of it. Secondly, I want to thank Prof. Prof. Dr. Thomas Noll for making himself available to be the secondary referee.

I give thanks to my supervisors Dipl.-Inform. Johanna Nellen and Dr. Martin R. Neuhäußer, who could always spare some of their time and who gave me helpful advice during the work on the subject.

Last but not least, I am especially grateful to my parents Dagmar and Ulrich Driessen for their continuous support throughout my studies in every respect. Without them, my master's course would not have been possible.

Contents

Chapter 1

Introduction

This master thesis deals with the verification of plant controls. The general idea behind the novel approach which is presented in this thesis is to analyze programs for programmable logic controllers (PLCs) by separating the discrete from the dynamic behaviour of the plant. PLC are often used to control plants by continuously executing a program thereby updating its input and output interfaces. Systems controlled by PLCs are usually safety-critical. These system might fail if specific states are reached. For example a tank which overflows might cause such a failure. A discrete analysis of such a program is fast and computes the exact states, which are reachable in the system. The analysis can be performed efficiently for large discrete systems. During the analysis of a PLC controlled plant the dynamic behaviour has to be considered as well. Combining the plant dynamics with the discrete behaviour might create large hybrid system, which are hard to verify. Thus, considering only the discrete behaviour, which describes a failed discrete analysis, i.e. a discrete counterexample, and combining it with the plant dynamics creates a smaller model. Hereby, a possible state space explosion, which results from combining the plant dynamics with the entire discrete behaviour, can be avoided. The goal is to use the hybrid analysis to refute discrete counterexamples, which can not occur due to the dynamic behaviour. Instead of analyzing large hybrid systems, we verify multiple smaller models in order to improve the overall verification and its runtime.

The general approach to verify PLC controlled plants is to analyze its control program disassociated from the model of the plant or analyze them in combination [HG98, ELS05, BCMP98]. The PLC program for instance can be analyzed by verifying a discrete model or timed automaton, if timed qualifiers are considered. Hybrid automata (HA) can be used to represent the plant dynamics. A parallel composition of the control program with the plant dynamics however might result in a large automaton. The counterexample guided abstraction refinement (CEGAR) techniques as presented in [ELS05] can be used to reduce the size of the automaton. Furthermore, the approach as to appear in [NA14] is able to construct smaller models of such systems by iteratively adding the plant dynamics. Thus, if a system is safe, the analysis might be able to verify a reduced model and as a consequence does not require to perform hybrid analysis on larger systems.

In this thesis we present a new approach which uses the result of a bounded model checking (BMC) [BCC+03] analysis for the control program to construct

a reduced model. We use BMC to verify the control program of the plant. The input language for the PLC program is a custom assembler-like instruction list called intermediate verification language. The same language is used to define the safety properties of the PLC. The BMC analyses the execution of multiple cycles of the PLC program using satisfiability modulo theories (SMT) solving. Moreover, it constructs a control flow automaton and is bounded by the search depth in the unrolled automaton. If the system is safe, there is no need for further analysis, however, the resulting path of a failed analysis due to unsafe states being reached is used as basis of models which are extended by the plant dynamics. The BMC analysis is able to store its current state, i.e. the SMT formulas representing the current control flow, when it detects a counterexample, thereby allowing the analysis to resume after excluding counterexamples, which have been refuted by the hybrid analysis.

The discrete counterexample path is combined with the dynamic behaviour of the plant and a hybrid automaton is constructed. Hybrid reachability analysis allows us to determine whether the discrete path described by the counterexample can still occur in the extended model where we additionally consider the dynamic behaviour. Existing tools which perform a hybrid reachability analysis [FLGD+11, CAS13] can be used to verify these models. If the extended model of the counterexample is confirmed, i.e. the discrete counterexample is reproduced in the hybrid analysis, a possible counterexample has been detected and the analysis stops with the result *unknown*. Due to the undecidability of the hybrid reachability analysis [ACH+95], the reachable states are approximated. Thus, the verification of the counterexample may occurred due to such approximation. Otherwise, the dynamic behaviour has prevented the hybrid reachability analysis from reproducing the counterexample. In this case, we determine the cause, i.e. why the counterexample was not reproduced, and use it to provide an explanation for the bounded model checker. This explanation is a prefix of BMC counterexamples, which can not be reproduced by the hybrid analysis due to the dynamic behaviour and can be used to exclude these counterexamples.

The counterexample prefix provided by the explanation is used to exclude the corresponding counterexamples and the BMC analysis is resumed. The stored state of the BMC is extended by formulas, which exclude the counterexamples with the given prefix. Thus, the collaboration of the BMC and the hybrid analysis allows us to iteratively exclude BMC counterexamples, which are not reproducible in the extended model. If no BMC counterexample is found, the given program is verified as no unsafe system states are reached for the given BMC bound. This collaborative verification approach is illustrated in Figure 1.1.

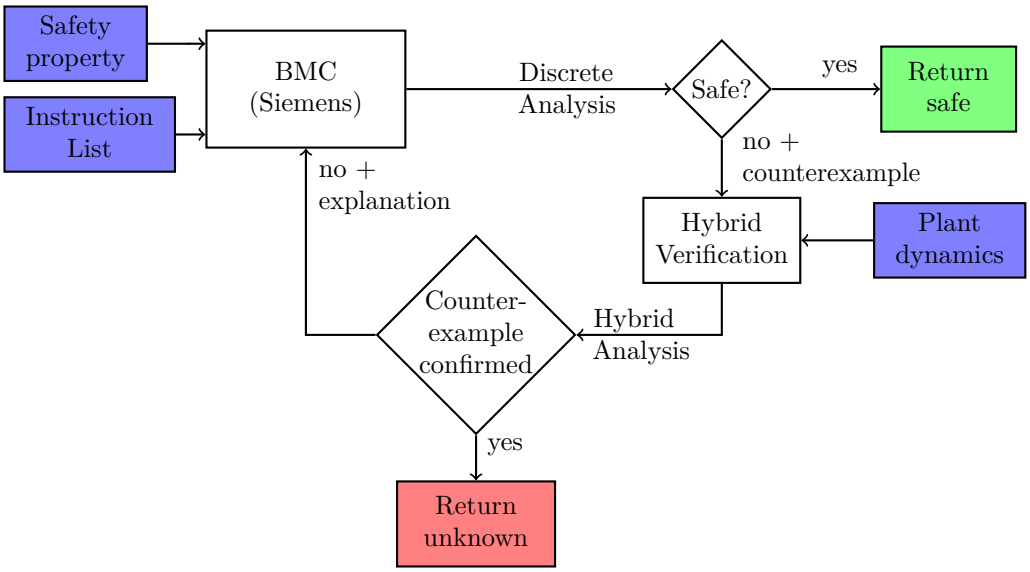

Figure 1.1: *Modular Verification of PLC-Controlled Plants*

In order to explain the basic requirements for our novel approach and, subsequently, elaborate on the approach itself, this master thesis is structured in the following way. In Chapter 2, we introduce the basic operation principle of PLCs and give an example for a tank system which is controlled by a PLC. Furthermore, we introduce a bounded model checker, which is currently being developed by the Siemens AG, to analyze the discrete behaviour and we introduce the counterexamples it produces. Moreover, we present hybrid automata to model the extended models for the discrete counterexamples as well as other third party tools, i.e. SpaceEx [FLGD+11] and Flow* [CAS13], for the analysis of hybrid automata. Additionally, we present a verification tool for PLC controlled plants, which provides functionality which can be repurposed for our approach. This tools provides us with representations and parsers for parts of the plant dynamics. We also introduce SMTInterpol [CHN12] as a means to check the satisfiability of logic formulas. Chapter 3 describes the transformation of BMC counterexamples into a hybrid automaton. We show how to construct an automaton for the counterexample, which contains all plant dynamics. Moreover, tools provided by the repurposed verification tool are used to construct suitable models for SpaceEx and Flow*. Additionally, we present an extension for BMC counterexamples, which increases their expressiveness. In Chapter 4 we introduce the parameters for the verification and discuss the settings for the hybrid reachability analysis. SpaceEx and Flow* have to be configured differently for each model. After a hybrid reachability analysis has been performed, in Chapter 5 we present methods to construct explanations from the outputs of SpaceEx and Flow*. We discuss the expressiveness of different explanations and elaborate on how the bounded model checker uses

these explanations to exclude certain counterexamples. Furthermore, we show how to extend the explanations to improve their expressiveness. In Chapter 6, the presented tank system and a second example are verified using our new approach. We discuss the results of these verifications present a runtime analysis. Moreover, we show where issues arise during the analysis and introduce improvements to the verification. Chapter 7 constitutes a final comment on the usability and the quality of the novel technique including an outlook regarding known issues and unexploited opportunities.

Chapter 2

Preliminaries

In this chapter we present the preliminaries, which are required in this thesis. The introduced components allow us to perform a discrete analysis, using bounded model checking, of a program of a programmable logic controller which controls a plant. Furthermore, we present hybrid automata and the architecture of a tool, which is able to verify sequential function charts, which can be used to program programmable logic controllers (PLCs), and the plant dynamics by combining them into a hybrid sequential function chart (HSFCs). Thereafter, these HSFCs are transformed into hybrid automata. Moreover, we introduce third-party tools to verify these hybrid automata and solve arising logic formula problems.

We introduce PLCs in Section 2.1, which are used to control plants. Their basic operation principle of PLC scan cycles is explained. We give an example of a system which can be controlled by a PLC in Section 2.2. This example consists of a single tank with a valve, which is used to fill the tank with water. Sensors provide information about the water level inside the tank. Furthermore, we present a method to perform bounded model checking (BMC) to verify the discrete behaviour, i.e. the control program of the PLC, of such a system in Section 2.3. The input of the BMC is the intermediate verification language (IVL), which is an assembler-like instruction list. Furthermore, the BMC employs satisfiability modulo theories (SMT) solving to verify the discrete behaviour. We introduce counterexamples, which define discrete paths that result in unsafe system states, for this BMC. Additionally, we discuss how the algorithm stores its current state when a counterexample is found and how it can resume its analysis from this state. An introduction to hybrid automata is given in Section 2.4 as we use these automata to construct a model for a BMC counterexample which contains the plant dynamics. We also present hybrid reachability analysis in Section 2.5 and introduce the third-party tools SpaceEx and Flow* to perform such an analysis. Moreover, we also describe the output needed to perform further analysis on the extended counterexamples. Some of the functionality to add plant dynamics to a discrete system, which allows us to extend BMC counterexamples with the dynamic behaviour, and to transform hybrid automata, which is required to create suitable models for SpaceEx and Flow*, are provided by the SFC Verification tool as presented in Section 2.6. Data structures to represent the plant dynamics are also provided by this tool. During the construction of the automaton, we use the SMT solver SMTInterpol as presented in Section 2.7 to check the satisfiability of logic formulas.

2.1 Programmable Logic Controllers

A Programmable Logic Controller (PLC) [BCMP98] is a digital computer which is used to evaluate digital and analogue sensors and interact with the outside world by operating devices like valves, pumps, motors and actuators through its outputs. E.g., a plant like a tank system can be controlled by a PLC where the inputs are provided by water level sensors and the outputs control pumps and valves within the system. The inputs and outputs of the PLC are built to be resistant to outside influences.

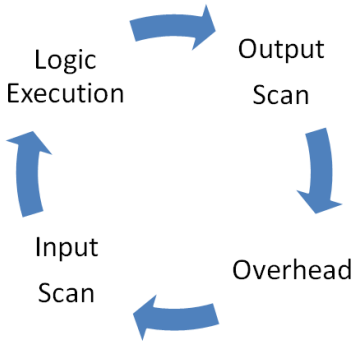

Figure 2.1: *PLC Scan Cycle*

In contrast to common computer programs, PLCs execute their programming cyclically. The details of the scan cycle vary between manufacturers, the most common scan cycles format however consist of four different phases as illustrated in Figure 2.1. The *overhead phase* contains maintenance and communications of which the former includes testing I/O integrity, verifying that the program logic did not change and checking if the program stopped responding using a watchdog timer, which times out if it is not reset periodically. The communication includes traffic over the PLC programmer port, I/O racks and other external devices. In the *input scan* phase a current snapshot of all digital and analog input values is taken. This snapshot is stored in the PLC's input memory table. The actual programming of the PLC is executed in the *logic execution* phase. Afterwards, the values resulting from the execution of the program are written to the output memory table. The content of the output memory table is written to the output modules in the *output scan* phase. The execution of this scan cycle is repeated until the PLC is shut down.

The scan cycle time is the time it takes the PLC to complete one scan cycle. This time usually ranges from a few hundred milliseconds on older and more complex PLCs to a few milliseconds on newer PLCs and PLCs with short and simple programming.

The international standard IEC 61131-3 [IEC07] defines 5 different languages to program PLCs. According to the standard function block diagrams (FBDs), ladder diagrams (LDs), structured texts (STs), instruction lists (ILs) or sequential function charts (SFCs) can be used to program PLCs. In our case, a type of assembler code, similar to instruction lists is used to program a PLC.

2.2 Tank System

In this thesis, we use a simple tank system as a running example for the verification process. The tank system consists of a single tank, which can be filled with water, and a valve, which controls the water flow inside the system. If the valve is open, the water level inside the tank is rising, while a closed valve allows the tank to drain through a hole in the tank. Furthermore, there are four sensors that indicate the water height inside the tank. The tank system is illustrated in Figure 2.2.

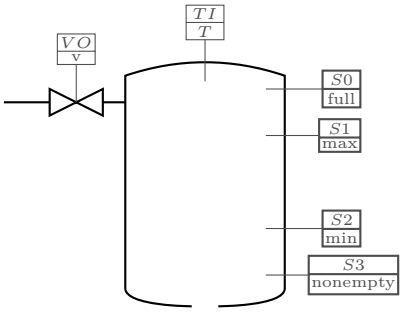

Figure 2.2: *Tank System*

The tank T has the four water height sensors *nonempty*, *min*, *max* and *full*, which each detect different water levels. If a sensor is active (returns *true*), the water height h in T is above the sensor position (specific height for h). These positions will be specified later. For now, we assume that if $full \Rightarrow max \wedge min \wedge nonempty$, $max \Rightarrow min \wedge nonempty$ and $min \Rightarrow nonempty$. Thus, if a sensor is active, all sensors with positions below are active as well. Valve v controls the water flowing into the tank system. Furthermore, we assume the incoming water flow when v is open is larger than the outgoing flow through the hole in the tank. The opening and closing of v depends on the sensor activity. If sensor *max* is activated, i.e., its state changes from *false* to *true*, v is closed. Accordingly, v is opened if the water height is too low, i.e if *min* is deactivated. This behaviour is supposed to keep the water at an appropriate level.

In order to ensure a safe system, T should never be too full or run dry. Hence, the sensors *full* and *nonempty* are used to define states in the system, which are unsafe and should not be reachable. In this example, if *full* is activated, then T is overflowing and if *nonempty* is deactivated, then T is running dry. Therefore, $full \vee \neg nonempty$ should never be satisfied.

In the following, we will refer to all input and output variables with matching prefixes. Thus, *nonempty*, *min*, *max* and *full* are now in_{nonempty}, in_{min}, in_{max} and in_{full} and v is renamed to out_v, making it easier to distinguish input and output variables.

2.3 Bounded Model Checking

In this section we present an algorithm which uses bounded model checking (BMC) to verify the discrete behaviour of a PLC program. Firstly, we describe the input model for the bounded model checking in Section 2.3.1. The bounded model checker we employ is presented in Section 2.3.2. Furthermore, we define a counterexample which the BMC algorithm uses to define an execution of a given system where an unsafe state is reached in Section 2.3.3.

2.3.1 Intermediate Verification Language

In this section we present the intermediate verification language (IVL) which is used to model the discrete behaviour of a PLC program. The language is based on Low Level Virtual Machine Intermediate Representation (LLVM-IR) [Lat02] without three-address code support. LLVM allows programs in arbitrary programming languages to be translated into the LLVM intermediate language. A similar concept is applied to the IVL.

For our purpose we directly define the programs as an assembler like code. Instructions supported by IVL include `assignment`, `goto`, `assume`, `assert`, `call` and `return`. Assumptions (`assume`) allow us to model the control programs behaviour, while assertions (`assert`) are used to define the safe states. The data types which can be used for variables are *Boolean*, *Integer* (Bit Vectors) as well as *Arrays* and *Records*. The variables can be defined globally or locally and the language provides the possibility to define functions. Writing a program for the tank system as described in Section 2.2 can be described as the code shown in Listing 2.1.

```
fun cycle(in_full : bool, in_max : bool, in_min : bool, in_nonempty : bool,
     result : bool*)
begin
  assert "no water at full sensor" !in_full
  assert "water at nonempty sensor" in_nonempty
  goto min, max, between

  min:
  assume (!in_min)
  set *result := true
  return

  max:
  assume (in_max)
  set *result := false
  return

  between:
  assume (in_min && !in_max)
  return
end

fun main()
vars
  in_full : bool, in_max : bool, in_min : bool, in_nonempty : bool, out_v :
       bool
begin
  head:
  set in_full := false
  set in_max := false
  set in_min := true
  set in_nonempty := true
  call cycle(in_full, in_max, in_min, in_nonempty, &out_v)
  goto head
  return
end
```

Listing 2.1: *Tank System IVL*

The program simulates the cyclic execution of a PLC program in the main function `main`. The label `head` at the beginning of the code allows the algorithm to return back to this position in the code using a `goto` command. The `set` commands only provide an initial assignment for the variables. During each PLC cycle, the variables assignments are generated by the BMC. The function `cycle` models the actual behaviour of the tank system, where the valve is opened and closed according to the sensor activity. The `goto min, max, between` branches the execution and performs the code after each label. A `return` is required after each code segment in order to stop each branch. Furthermore, the `assert` commands define the conditions the system has to satisfy. If an assertion is violated, the system is an unsafe state.

2.3.2 Bounded Model Checking Algorithm

Model checking examines if a property is satisfied in a model of a given system. This property can be a set of forbidden states, which should not be reachable in the system. The general idea is to find an execution which reaches one of these forbidden states, thus proving the model to be incorrect. Finding such a counterexample allows further analysis to determine the problem in the current model. These counterexamples however can be long and difficult to analyze as the PLC runs in a loop until it is shut down. For our purposes we employ bounded model checking to reduce the number of states which are examined while analyzing discrete systems.

Bounded model checking is initialized with a parameter k, which restricts the path length of the analysis as well as the length of the counterexamples. Due to this restriction, bounded model checking does not require exponential space and models can usually be checked very fast. The performance decreases if we have a big system and a large parameter k. In general, bounded model checking is not complete as the procedure might not detect longer counterexamples.

For our purposes, we employ a bounded model checker, which uses satisfiability modulo theories (SMT) solving to verify a model similar to the approach proposed in [CK03]. A program modeled by IVL is transformed into a control flow automaton (CFA) by mapping a bit-precise memory model onto the automaton. The CFA uses the guarded command language (GCL) [Dij97] to describe the flow of the PLC program. The parameter k in this case refers to the maximum depth, which is analyzed in the unrolled automaton. Afterwards the CFA is transformed by replacing the GCL sequences with first order logic formulas. The BMC unrolls the automaton to create sequences of formulas, which are used to determine a set of constraints C and a set of properties P [CK03]. The constraints and properties are computed for each step in the automaton, while the BMC verifies that $C \Rightarrow P$ is valid in each step. If this is not the case, a counterexample is generated.

The general architecture of the tool we use also provides a CEGAR approach for the analyis. Furthermore, a third engine (IC3) is planned to be integrated. However, we only consider the BMC engine for the analysis. Figure 2.3 illustrates the general architecture of the verification tool.

The BMC currently uses an IVL parser to parse the IVL code as presented in Section 2.3.1. Afterwards, this code is transformed in an internal representation of the IVL. Thereafter, the IVL is transformed into a bit-precise memory model before the control flow automaton is constructed. BMC is then used to verify the program and provide a counterexample if the system is unsafe.

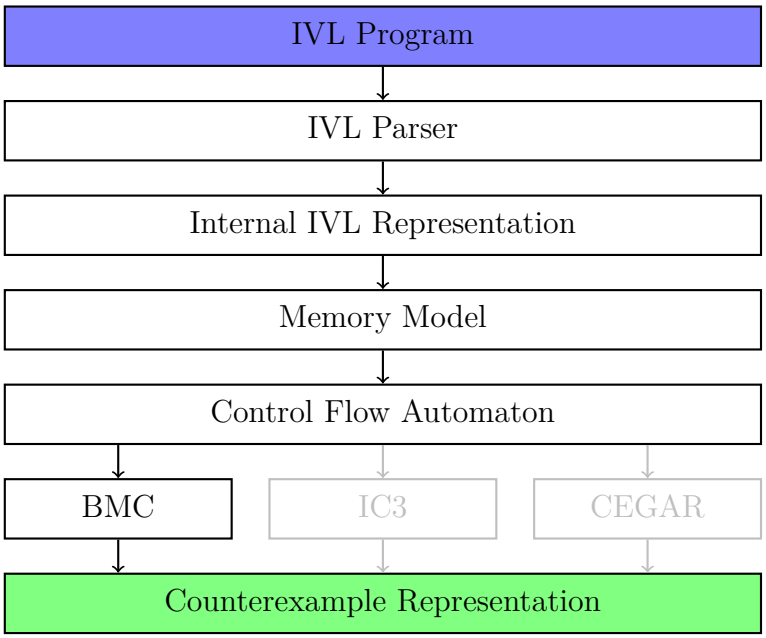

Figure 2.3: *BMC Architecture*

Halting the BMC after the detection of a counterexample, allows the us to analyze this counterexample further. If the BMC finds a counterexample, the current states is saved so the analysis can continue from this state. This can be achieved by utilizing the SMT solver functionality. The BMC uses Z3 [DMB08] to solve the occurring SMT formulas. The theory employed for the SMT solving is quantifier-free expressions over *Booleans*, *Arrays* and *BitVectors*, where all array index and value sorts are bit vector sorts. The stored state is the state before the counterexample has been found, thus allowing us to perform further analysis on the counterexample and to exclude the counterexample from the BMC analysis if necessary.

This exclusion is accomplished by the BMC restoring the state when the last counterexample was detected and adding a new formula to the SMT solver. This new formula contains restrictions, which preclude the sequence of values that describe the counterexample from occurring again in this analysis. Thus, it is possible to use a discrete counterexample provided by the BMC as an origin for an analysis of the dynamic behaviour. The results of this hybrid analysis can be used to verify the validity of the counterexample with given plant dynamics. Therefore, the counterexamples that have been disproved by the hybrid analysis can be excluded from the next iteration of the BMC as described before.

The presented BMC is still a work in progress and has some restrictions in regard to its functionality. The previously mentioned memory model which is used during the transformations is quite naive. All operations are modeled

correctly but there is still room for improvements. Furthermore, the IVL can still be optimized by functionality like slicing, constant-propagation and forward-expression substitution. The BMC algorithm currently does not support non-determinism in the automaton or floating-points. All these problems are currently being worked on.

2.3.3 Bounded Model Checking Counterexample

The counterexamples generated by the bounded model checker, which we employ, consist of an execution path which can be transformed into sequences of variable valuations and are produced if an assertion is violated. The structure of a counterexample is defined as shown in Definition 2.1.

Definition 2.1 (Counterexample)
A counterexample $(Vars, Seqs)$ *defines a set of variables Vars and their valuation sequences Seqs.*

- $Vars := \{var_1, \ldots, var_m\}$ *is a finite list of discrete variables*
- $Seqs := \{s_1, s_2, \ldots, s_m\}$ *is a set of sequences* $s_i := (v_{i,1}, v_{i,2}, \ldots, v_{i,n})$ *of* $var_i \in Vars$ *with length n and variable valuations* $v_{i,1}, v_{i,2}, \ldots, v_{i,n}$ *for each variable* var_i *with* $i \in \{1, 2, \ldots, m\}$

The counterexample describes a path of discrete valuations in the discrete system, which ends in an unsafe state. A list of variables and their respective types are defined in the header of the counterexample. Currently the bounded model checker supports five different data types (boolean, signed and unsigned bit vectors of sizes 16 and 32). Currently, we restrict our analysis to boolean variables as the BMC is currently not able to process integer values. Furthermore, all variables in our approach either have the prefix `in_` or `out_`, which determines if it is an input or an output variable, respectively. All consequent lines define an valuation sequence for a variable, where each valuation $v_{i,j}$ of variable $var_i \in Vars$ correlates to a PLC cycle j. The order of sequences corresponds to the order in the variable list.

We assume an exemplary counterexample for the tank example discussed in Section 2.2. The counterexample provides a sequence of valuations for variables corresponding to the sensors and the valve of the tank system, which ends in

an unsafe state, where in_{full} is active. Thus, the tank is overflowing in the last PLC cycle. The counterexample is presented in Listing 2.2.

```
(in_full:bool,in_max:bool,in_min:bool,in_nonempty:bool,out_v:bool)
(0,0,0,0,0,0,0,0,1)
(0,0,0,0,0,0,0,1,1)
(1,1,0,0,0,1,1,1,1)
(1,1,1,1,1,1,1,1,1)
(0,0,0,1,1,1,1,1,1)
```

Listing 2.2: *BMC Counterexample*

The counterexample given in Listing 2.2 provides a variable and data type declaration for each variable as well as different assignment sequences for the sensors and the valve. These sequences describe a water level, which decreases, as the valve is closed, until sensor in_{min} is no longer active. At this point the valve is opened and the water level quickly rises until in_{full} is active, thus reaching an unsafe state.

2.4 Hybrid Automata

Hybrid automata (HA) [ACH+95] are automata, which can model discrete as well as continuous behavior of a system. We use them to build a hybrid model from a discrete counterexample of the BMC as presented in Section 2.3.3 and the dynamic behaviour of the plant. A reachability analysis on this model reveals whether we can refute or confirm a discrete counterexample. A formalization of a hybrid automata (HA) is given in Definition 2.2.

A hybrid automata consists of locations, which include flows and invariants. The flows describe how the continuous variables evolve over time. Each invariant of a location must be satisfied for the system to remain in the location. The guard and assignments of a transition between two locations are specified by a transition relation $\mu \subseteq V \times V$. The transition can be taken with an evaluation v changing the evaluation to ν' iff $(\nu, \nu') \in \mu$. Furthermore, the invariant of a location the system is about to enter via a transition must be satisfied after the assignments are applied. A transition (l, a, μ, l') can only be taken, if its guard as defined by the transition relation μ and the source locations invariant are satisfied by the current variable valuation $\nu \in Inv(l)$ and the new variables valuations, which are updated according to the assignments of the transition as defined by μ, afterward satisfy the target locations invariant $\nu' \in Inv(l')$. The special properties of τ-transitions are only relevant during the parallel composition of hybrid automata.

Definition 2.2 (Hybrid Automaton)
A Hybrid Automaton is a tuple $\mathcal{H} := (Loc, Var, Lab, Trans, Flow, Inv, Init)$ *where*

- Loc *is a finite set of locations*
- $Var := Var_d \cup Var_c$ *is a finite set of discrete and continuous variables; A valuation* $\nu \in V, \nu : Var \to \mathbb{R}$ *assigns a value to a variable where* V *is the set of all valuations* $\nu : Var \to \mathbb{R}$.
- Lab *is set of synchronization labels*
- $Trans \subseteq Loc \times (Lab, 2^{V^2}) \times Loc$ *is a set of labeled transitions including* τ*-transitions* (l, τ, Id, l) *for each location* $l \in Loc$ *with* $Id = \{(\nu, \nu) | \nu \in V\}$ *are* τ*-transitions.*
- $Flow$ *is a function assigning a set of time-invariant flows* $f : \mathbb{R}_{\geq 0} \to V$ *to each location, i.e.,* $\forall m \in Loc : f \in Flow(m) \implies (f + t) \in Flow(m)$ *where* $(f + t)(t') = f(t + t')$ *for all* $t, t' \in \mathbb{R}_{\geq 0}$*;*
- $Inv : Loc \to 2^V$ *is a function that assigns an invariant to each location*
- $Init \subseteq Loc \times V$ *is a set of initial states. A pair of a location and valuation is called a state*

There are two different kinds of steps in a hybrid automaton. The discrete step allows the automaton to change from one location into another using a transition. For a system to change locations, a transition between the locations is required and its guard and its source locations invariant have to be satisfied. Furthermore, the target locations invariant has to be satisfied after updating the variables according to the assignments of the transition. Assuming locations $l, l' \in Loc$ and transition relation $\mu \subseteq V \times V$ and label a, the discrete step is described in Equation (2.1).

$$\frac{(l, a, \mu, l') \in Trans \quad (\nu, \nu') \in \mu \quad \nu \in Inv(l) \quad \nu' \in Inv(l')}{(l, \nu) \xrightarrow{a} (l', \nu')} \tag{2.1}$$

Hybrid automata can also perform time steps. These steps represent the passing of time and evolution of continuous variables in a system. During such a time elapse, the hybrid automaton stays in the same location and updates the variables according to the flow function. During this time step the

invariant of the current location must not be violated. For a location $l \in Loc$, variable valuations $v, v' \in V$ and a time horizon t the time step is defined in Equation (2.2).

$$\frac{f \in Flow(l) \quad f(0) = \nu \quad f(t) = \nu' \quad t \geq 0 \quad \forall 0 \leq t' \leq t : f(t') \subseteq Inv(l)}{(l, \nu) \xrightarrow{t} (l, \nu')} \tag{2.2}$$

A problem which can occur in hybrid automaton is Zeno behaviour. Zeno behaviour describes the case where the system can take an infinite amount of discrete jumps in a finite amount of time. This behaviour is not a problem in the actual system, but may occur due to the model abstraction. If neither a discrete nor a time step can be taken in the current location and variable valuation, the hybrid system is in a deadlock.

Figure 2.4 is a simple hybrid automaton with a single continuous variable d, which we use to introduce the notations of the hybrid automaton.

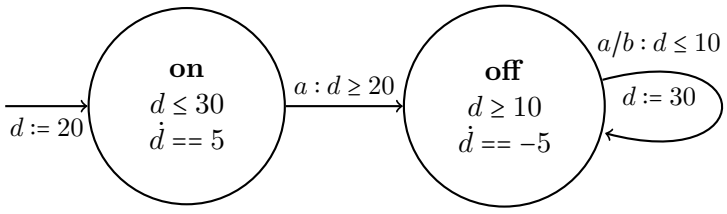

Figure 2.4: *Exemplary Hybrid Automaton*

The system has one continuous variables, i.e. the set of variables is defined as $Var = Var_c \cup Var_d$ where $Var_c = \{d\}$ and $Var_d = \varnothing$ and two locations **on** and **off**, i.e a set of locations $Loc = \{\textbf{on}, \textbf{off}\}$, with invariants $d \leq 30$ and $d \geq 10$ respectively. The invariants for the locations are defined as $Inv(\textbf{on}) = \{\nu \in V | \nu(d) \leq 30\}$ and $Inv(\textbf{off}) = \{\nu \in V | \nu(d) \geq 10\}$. Furthermore, the flows defined for these locations are $\dot{d} == 5$ for **on** and $\dot{d} == -5$ for **off**. The flows are shown in Equations (2.3) and (2.4).

$$Flow(\textbf{on}) = \{f : \mathbb{R}_{\geq 0} \to V | \exists c \in \mathbb{R}. \forall t \in \mathbb{R}_{\geq 0}. f(t)(d) = 5t + c\} \tag{2.3}$$
$$Flow(\textbf{off}) = \{f : \mathbb{R}_{\geq 0} \to V | \exists c \in \mathbb{R}. \forall t \in \mathbb{R}_{\geq 0}. f(t)(d) = -5t + c\} \tag{2.4}$$

The set of synchronization labels is defined as $Lab = \{a, b\}$. The transition from **on** to **off** labeled with a has a guard $d \geq 20$ and no assignments, while the self

loop of location **off** labeled with a/b has a guard $d \leq 10$ and the assignment $d := 30$. The transition with label a/b describes two transitions where one transition is labeled a and the other is labeled b, while they have the same source and target location as well as the same guard $d \leq 10$ and assignment $d := 30$. The set of transitions is defined in Equations (2.5) to (2.7) where $Trans_\tau$ is the set of all τ-transitions.

$$Trans = \{(\mathbf{on}, a, \{(\nu, \nu') \in V^2 | \nu(d) \geq 20 \wedge \nu(d) = \nu'(d)\}, \mathbf{off}), \tag{2.5}$$
$$(\mathbf{off}, a, \{(\nu, \nu') \in V^2 | \nu(d) \leq 10 \wedge \nu'(d) = 30\}, \mathbf{off}), \tag{2.6}$$
$$(\mathbf{off}, b, \{(\nu, \nu') \in V^2 | \nu(d) \leq 10 \wedge \nu'(d) = 30\}, \mathbf{off})\} \cup Trans_\tau \tag{2.7}$$

The only initial state of the hybrid automaton is the state with initial location **on** and variable assignment $d := 20$. Thus, the initial states are defined as $Init = \{(\mathbf{on}, \{\nu \in V | \nu(d) = 20\})\}$

Furthermore, we define a path in a hybrid automaton in Definition 2.3 as we are interested in which states can be reached during the analysis of a hybrid automaton.

Definition 2.3 (Path)
An (automaton) path ρ is a finite sequence of states, which consist of a location and variable valuation, which describes a possible path in an automaton

- $\rho = (loc_1, V_1) \rightarrow (loc_2, V_2) \rightarrow \ldots \rightarrow (loc_n, V_n)$ *describes an automaton path of length n where (loc_i, V_i) is the state at position i in the path with location loc_i and is a set of variable valuation V_i, where (loc_1, V_1) is an inital state.*

These path can be determined by a hybrid reachability analysis. For our purposes, we reduce a path $\rho = (loc_1, V_1) \rightarrow (loc_2, V_2) \rightarrow \ldots \rightarrow (loc_n, V_n)$ to a *location path* $loc_1 \rightarrow loc_2 \rightarrow \ldots \rightarrow loc_n$, if we only need to consider the reachable locations of the automaton.

Furthermore, we differentiate between linear and non-linear hybrid automaton. Linear hybrid automata are hybrid automata, where all flows, invariants and transition relations are defined by linear expressions [ACH+95]. Non-linear hybrid automata are not restricted in this way.

2.5 Hybrid Automaton Reachability

The reachability problem for hybrid automaton is to decide whether there is a path from an initial state to a specific target state. It is an important problem for verifying automata. Unfortunately, for hybrid automata it is undecidable in general [ACH+95]. To verify hybrid automata we need to compute the reachable states of the automata and check if the system stays in safe states, i.e., never reaches forbidden/bad states. These forbidden states define the safety conditions for a given system. Using approximation, the reachable states of the hybrid model can be computed efficiently.

In order to verify an automaton given a set of forbidden states either a forward or backward analysis is performed. During the forward analysis, the verification starts computing the reachability of the initial states. If the reachable set of states intersects with the forbidden states, the system is not safe under the given safety conditions. The backwards analysis performs a backwards reachability starting from the forbidden states. If the intersection of the backwards reachable states and the initial states is not empty, the system is not safe. For either approach the reachability of a hybrid automaton has to be computed. These computations terminate when either the specified maximal number of discrete and time steps has been performed or a fixed point of the state set is found, i.e., a point where the time and jump successors of all reachable states have been computed without finding a new state.

Since the reachability for hybrid automata is undecidable in general as mentioned before, the most common approach to analyze hybrid systems is to approximate the reachable states in each step to determine the reachable states of the automaton [ACH+95]. For our purposes, we use the SpaceEx tool platform in order to perform a reachability analysis [FLGD+11]. We also utilize Flow* [CAS13] to perform forward reachability analysis using Taylor Model flowpipes for hybrid automata.

The third-party tools are introduced in the following sections. A detailed description of the SpaceEx tool platform is given in Section 2.5.1. Furthermore, Flow* is presented in Section 2.5.2.

2.5.1 SpaceEx

SpaceEx [FLGD+11] can perform a reachability analysis on hybrid systems as well as a safety analysis. The models for SpaceEx are specified using the SpaceEx modeling language [SC10]. The analysis supports different algorithms, which include PHAVer [Fre05], the LGG Support Function and the STC en-

hancement for LGG scenario. The PHAVer algorithm is applicable on linear hybrid automata, i.e. hybrid systems with piecewise constant bounds on the derivatives, and produces precise results for the reachability. During its execution, it computes exact results for piecewise constant flows. The LGG Support Function scenario implements a variant of the Le Guernic-Girad (LGG) algorithm [GG09]. During the reachability analysis, support functions are used to approximate the reachable states. The STC scenario is based on LGG and computes less convex sets and more precise images of the discrete transitions. These scenarios over-approximate the reachable states and can be applied to hybrid systems with piecewise affine dynamics with non-deterministic inputs. SpaceEx requires the hybrid automaton being stored in a XML-File and a configuration file specifying initial states, output formats, number of iterations, where an iteration describes the computations of all reachable states from a given state and time horizon, and other preferences. The configuration (CFG) file can also be used to define forbidden states.

The number of iterations defines the amount of states for which the reachable states are computed given a time horizon. This option is required as the reachability analysis for hybrid automata is undecidable in general and might not reach a fix point, which would cause the analysis to terminate. If -1 is set as the iteration number, SpaceEx will only terminate if a fix point is found. If the system does not exhibit Zeno behaviour, we can use this option due to the sequential structure of the counterexample which translates to the automaton as well the automaton being a finite model and the only reachable path being loop free. We can also find fix points in a system with Zeno behaviour, however the analysis might require many iterations to detect such fix points. Furthermore, it is possible to set the range of the time steps. If the SpaceEx configurations are set too conservative, the tool might produce meaningless results, as only few reachable states are computed.

SpaceEx produces several different outputs. The INTV output generates an interval file containing over-approximated intervals for each variable containing its reachable values. These intervals are given for the entire automaton and for each location separately. Another output file is the TXT-File, which provides information about the state and vertices of the polytopes that model the reachable areas of the continuous variables. The state information include initial values and locations. These outputs allow us to determine the reachable states, but do not provide information about the paths which are taken during the reachability analysis.

The third textual output is a non-standard console output. SpaceEx has been modified to provide additional console output. When SpaceEx is executed using debug level 2 as verbosity, information on the computation of the time and

discrete steps is given. Using a slightly modified version of SpaceEx, enough information is provided to determine the set of paths which have been analyzed. In order to perform our analysis of the hybrid reachability, we require this extended console output.

In addition to the textual outputs of SpaceEx, the tool platform also provides a variety of graphical output formats. Figures 2.5 and 2.6 show exemplary two and three dimensional outputs of SpaceEx.

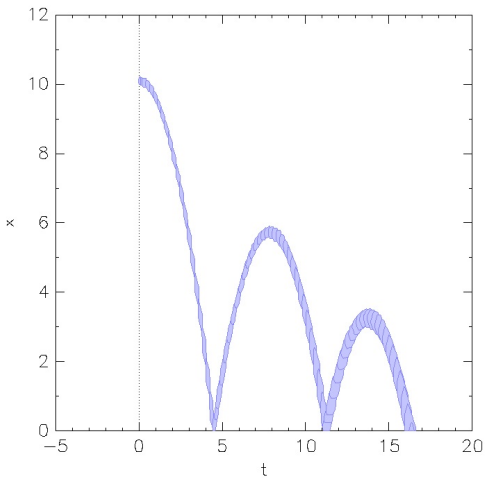

Figure 2.5: *2d SpaceEx Output*

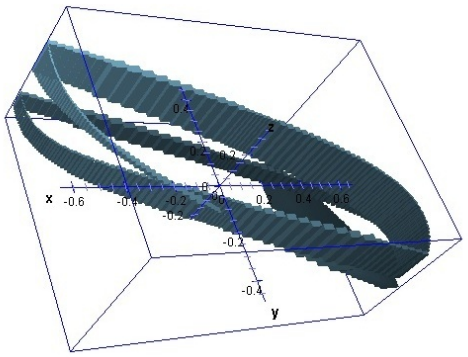

Figure 2.6: *3d SpaceEx Output*

Graphical outputs supported by SpaceEx are the GEN (Vertice List) and JVX output [JP00]. Both of these formats can be visualized using third-party tools. *Graph* of the *Plotutils* [MT00] package can be used for GEN and *JavaView* [Pol06] for the JVX format.

SpaceEx also has some restrictions. SpaceEx can handle linear convex guards and invariants, i.e. it does not support the strict inequality operators $<$ and $>$. This can be solved by overapproximation to $\leq$ and $\geq$ or by approximating with a deviation ϵ. Disjunction is not allowed in models, but occurrences of disjunction in transition guards or invariants can be split into new transitions and locations respectively, after they have been transformed into disjunctive normal form. Furthermore, the supported variable types only include *integer* and *real*. Boolean variables are redefined as numerical variables where the boolean conditions are transformed into equality checks to the values of 1 for *true* and 0 for *false*.

2.5.2 Flow*

The Flow* tool [CAS13] allows to analyze hybrid automata by generating flowpipes using Taylor Models (TMs) [BM98]. Flowpipes over-approximate the reachable states over a certain time horizon. The previously mentioned TMs were originally proposed by Berz and Makino in order to represent functions by Taylor polynomials. The flow of the continuous variables of a system can be constrained accurately by Taylor models as they are a sets defined by polynomials over intervals, which are bloated by an additional interval. Furthermore, the usage of Taylor models allows Flow* to analyze non-linear hybrid systems [CAS12].

Flow* is able to compute Taylor model flowpipes for continuous systems which are described by non linear ordinary differential equations (ODEs). Furthermore, Flow* supports the computation of the continuous evolution through ODEs with discrete jumps. The verification tool is able to handle dynamics represented by non-linear ODEs, which may include non-polynomial terms such as sine, cosine and square root. Additionally, Flow* supports location invariant and transition guards described by the conjunction of polynomial constraints. Transition assignments are defined by polynomial mappings with uncertainties.

The tool also provides several output formats for a reachability analysis. The FLOW file contains the reachable states of the system in the form of flowpipes, while the PLT file offers a graphical plot format, which can be used to visualize the output. Furthermore, if forbidden states are defined and reachable, Flow* constructs a counterexample file, which contains all paths from an initial state to a forbidden state.

Flow* has similar restrictions as SpaceEx. Disjunctions in transition guards and location invariants have to be split into multiple transitions and locations containing only conjunctions of polynomial constraints. On the other hand, the current version of Flow* star is restricted to initial states which share the same initial location. This restriction can be resolved by creating a new initial location incorporating all original initial states and constructing transitions to the original initial locations. These transitions are guarded with the original initial assignments for each original initial location. Thus, after the reachability analysis has taken such a transition the new state corresponds to the original initial state of the automaton.

Unfortunately, the third party GSL matrix library [Gou09] which is used for Flow* computations is the origin of a matrix rank deficiency exception. This exception occurs in some more complex systems and until fixed prevents us from using Flow* as a verification tool. However, we already have a functional architecture, which allows us to utilize Flow* for the hybrid reachability analysis as soon as the problem has been solved.

2.5.3 Reachable Paths

Neither SpaceEx nor Flow* provides us directly with the output which is required for our purposes. We can however derive such an output from the given outputs of both tools. We are interested in the paths, which the hybrid system visits during its analysis. The required outputs are reachability trees as they provide a suitable data structure to store the reachable paths. We show how to derive a reachability tree from the SpaceEx and Flow* output later on.

We use reachability trees to represent these paths as defined in Definition 2.3, so that a reachability tree contains all states which have been reached during the analysis. A definition for a reachability tree is given in Definition 2.4.

Definition 2.4 (Reachability Tree)
Given a hybrid automaton $\mathcal{H} = (Loc, Var, Lab, Trans, Flow, Inv, Init)$*, a reachability tree is a connected cycle-free graph containing the states reachable during the hybrid reachability analysis. Each node* $\eta = (loc, V, C)$*, which represents a reachable state, consists of:*

- $loc \in Loc$ *is the location of the automaton the analysis has reached*
- V *is a set of variable valuations of the variables in* Var
- C *are a set of successor nodes with states that are direct successors of the state* (loc, V)

The root node $\eta_r = (loc_r, V_r, C_r)$ *corresponds to an initial state of* $\mathcal{H}$ *with* $(loc_r, V_r) \in Inv$*. In a leaf node* $\eta_l = (loc_l, V_l, \varnothing)$ *either there are no more states reachable from* (loc_l, V_l) *or the reachability analysis has been restricted.*

The tree can be used to determine the reachable locations in a hybrid automaton. If the analyzed automaton has multiple initial states, there might be multiple root nodes, thus instead of a tree, the output corresponds to a forest. Figure 2.7 shows an exemplary reachability tree, where each node contains a location loc_i and a set of variable valuations V_i.

For our purposes, we only consider the paths that start in the root node (loc_1, V_1) and end in one of the leaf nodes since we are interested in the complete paths. All remaining path fragments, which are included in these paths are ignored as they provide redundant information. The relevant paths of the tree in Figure 2.7 are shown in Equations (2.8) to (2.11)

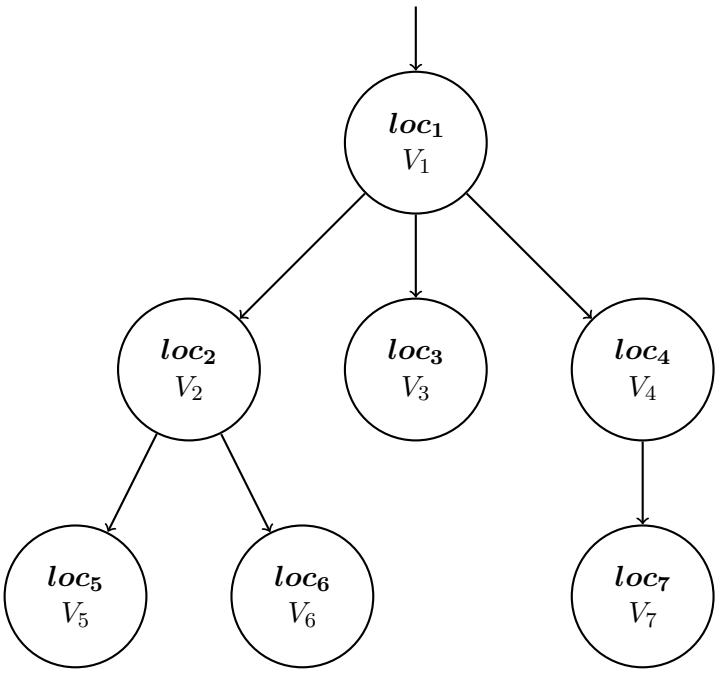

Figure 2.7: *Reachability Tree*

$$(loc_1, V_1) \rightarrow (loc_3, V_3) \tag{2.8}$$
$$(loc_1, V_1) \rightarrow (loc_2, V_2) \rightarrow (loc_5, V_5) \tag{2.9}$$
$$(loc_1, V_1) \rightarrow (loc_2, V_2) \rightarrow (loc_6, V_6) \tag{2.10}$$
$$(loc_1, V_1) \rightarrow (loc_4, V_4) \rightarrow (loc_7, V_7) \tag{2.11}$$

The resulting list of paths in Equations (2.8) to (2.11) contains all states, which have been reached during the hybrid reachability analysis.

2.6 SFC Verification Tool

We integrated the hybrid analysis of BMC counterexamples into an existing tool for SFC Verification. The SFC Verification tool provides functionality to transform a SFC and given plant dynamics, which model the dynamic behaviour of a plant, into a hybrid automaton, thus allowing the user to verify a PLC program using hybrid verification tools [NA14]. The dynamic behaviour of a plant is provided using conditional ordinary differential equations as shown in Section 2.6.1. Using these plant dynamics the given program is verified as described in Section 2.6.2.

2.6.1 Conditional Ordinary Differential Equations

The dynamic behaviour of a plant can be specified by conditional ordinary differential equation systems (conditional ODEs) [NA12]. We define conditional ODEs in Definition 2.5

Definition 2.5 (Conditional ODE System)
Let ODE_{Var} be the set of all ordinary differential equations over Var and Conds the set of all conditions over Var. A conditional ODE system is a pair $c := (cond : ODEs)$ with:

- *$cond \in Conds$ is a condition over the variables Var*
- *$ODEs \subseteq ODE_{Var}$ is a set of ODEs over Var*

The condition determines when the related set of ordinary differential equation systems is used to describe the flow of continuous variables. A list of conditional ODEs as depicted in Equations (2.12) to (2.14) can be used to define the dynamic behavior of the hybrid system.

$$cond_1 : ODEs_1 \tag{2.12}$$
$$cond_2 : ODEs_2 \tag{2.13}$$
$$\vdots \quad : \quad \vdots$$
$$cond_l : ODEs_l \tag{2.14}$$

Some conditions $cond_i$ for $i \in \{1, 2, \ldots, l\}$ overlap at the boundaries, which might cause Zeno behaviour in the hybrid model as the hybrid model is able to switch between the overlapping conditional ODE systems. This might result in the system switching conditional ODEs after each computation of reachable states.

Assuming a tank system where the inflow is controlled by a valve and the outflow occurs due to a leak in the tank, where out_v defines the current state of the valve and the water height is h, we use a list of conditional ODE systems to model the height change of the water inside a tank. The change of h depends on whether the valve is open $out_v == 1$ or the valve is closed $out_v == 0$. A list of conditional ODE systems for this tank system can be defined as shown in Equations (2.15) and (2.16)

$$out_v == 1 : \dot{h}_1 = 2, \tag{2.15}$$
$$out_v == 0 : \dot{h}_1 = -2 \tag{2.16}$$

As described in Section 2.2, an open valve corresponds to the water flowing into the tank being larger than the amount of the water flowing out through the hole in the bottom of the tank. This results in the water level rising by the value of 2 for each time unit. Furthermore, a closed value corresponds to the water level being reduced by 2, i.e. $\dot{h}_1 = -2$, for each time unit, due to water leaking out through a hole in the tank.

2.6.2 SFC Verification

The SFC Verification allows the verification of PLC-controlled plants. The verification parses a SFC and the plant dynamics given by conditional ODEs and constructs a hybrid sequential function chart (HSFC), which retains the discrete behaviour of the SFC and incorporates the plant dynamics. The HSFC is transformed into a hybrid automaton [NA12]. The SFC Verification tool uses a toolchain to perform this transformation.

This toolchain provides several tools to transform a hybrid automaton as the hybrid automaton resulting from the HSFC transformation might not provide a suitable model for SpaceEx or Flow*. We repurpose these tools to transform the automata we obtain during our analysis. These tools allow us to remove some of the unreachable locations and their transitions as well as initial states which contain unsatisfiable invariants in their initial location. Tools to split the disjunction in transition guard and location invariants are also provided. Fixes for several other problems pertaining to the restrictions of SpaceEx and Flow* are made available by the tools.

We use some of these tools to transform our hybrid automaton in a verifiable model. Furthermore, we utilize the model generators provided by the SFC verification tool. These generators allow us to transform a HA into a valid tool model for SpaceEx using the SpaceEx modeling language [SC10] or a model using the Flow* syntax. Once such a model is generated by the tool, we can use SpaceEx or Flow* to verify the given system.

The interface of the SFC Verification also provides a general configuration. Using these configurations it is possible to set paths to verification tool executables. Furthermore, the general configuration file determines which tool is being used for the reachability analysis. Listing 2.3 illustrates an excerpt of an exemplary general configuration file containing the settings relevant for our purposes.

Using all the functionality provided by the SFC Verification tool, we are able to transform a hybrid automaton in suitable models for SpaceEx and Flow* and construct the models files for the verification tools with the required syntax.

```
SpaceEx = /home/.../spaceex_exe/spaceex
FlowStar = /home/.../flowstar-1.2.0/flowstar

Verification Tool = SpaceEx
...
```

Listing 2.3: *Property File*

2.7 SMTInterpol

During the construction and analysis of the hybrid automata, we employ SMTInterpol for arising satisfiability problems. As our tool is developed in Java we have decided to use SMTInterpol [CHN12], as it provides an easy interface for our purposes in the form of a Java API. Moreover, SMTInterpol has achieved high rankings in the SMT competition SMT-COMP 2014. SMTInterpol is an interpolating SMT solver which supports the quantifier-free fragments of the combination of the theory of uninterpreted function as well as a theory of linear arithmetic over integers and reals. Furthermore, SMTInterpol is SMTLIB 2 [BST10] compliant, which is an common standard for SMT solvers.

SMTInterpol converts asserted formulas into conjunctive normal form, which is a conjunction of disjunction of literals, before solving them. Moreover, it roughly follows the Davis–Putnam–Logemann–Loveland (DPLL) [GHN+04] as an underlying algorithm. There are two different solver, which are used for satisfiability checking. The first solver is able the handle uninterpreted functions while the second solver is applicable to linear arithmetic. For our purposes, we restrict the satisfiability analysis to linear arithmetic, as we do not use uninterpreted functions in our models.

SMTInterpol can produce models for satisfiable formulas as well as provide resolution proofs for unsatisfiable formulas.

2.8 Summary

In this chapter we have introduced programmable logic controllers which control plants. An example for such a plant is given in the form of a tank which can be filled with water or drained. The relevant water levels in this tank can be determined by sensors. The discrete senor activity is verified by the BMC we presented as it is able to detect if unsafe states are reachable in a program of a PLC. Furthermore, counterexamples are defined, which are produced if such an unsafe state is reached. Hybrid automata are used to represent

the BMC counterexamples that have been extended with the plant dynamics. Constructing hybrid automata to describe the counterexamples allows us to perform hybrid reachability analysis using third-party tools like SpaceEx and Flow*. Moreover, we are able to repurpose the existing architecture of the SFC Verification tool, which already provides functionality to parse parts of the plant dynamics as well as an automaton data structure and automaton transformation tools. We use these to construct suitable models for SpaceEx and Flow*. During some transformations we need to solve satisfiability problems for logic formulas. We employ SMTInterpol for these tasks when they arise.

Chapter 3

Hybrid Model

In this chapter we present how to transform a bounded model checking counterexample as defined in Section 2.3.3 to construct a hybrid automaton, which combines the discrete behaviour provided by the counterexample with the plant dynamics. The construction of such a hybrid automaton is accomplished by adding the dynamic behaviour to a sequential automaton describing the different discrete assignments for each PLC cycle, which are defined by a counterexample. Once the automaton has been constructed, we can use hybrid reachability analysis to verify the extended dynamic model describing the counterexample. Figure 3.1 depicts the section of the complete verification procedure which is discussed in this chapter.

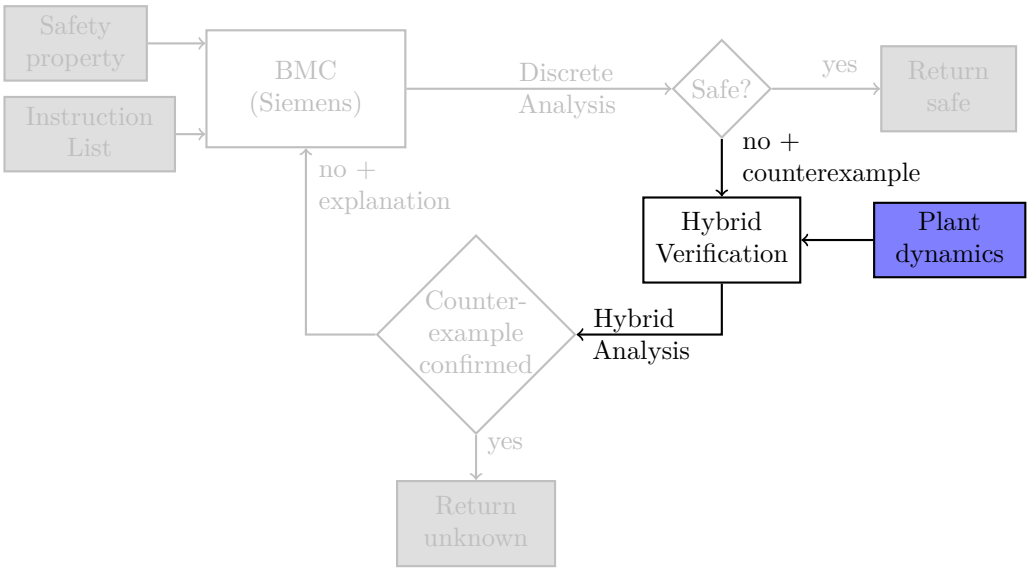

Figure 3.1: *Automaton Generation*

In Section 3.1 we show how we can construct a hybrid automaton which represents the discrete behaviour of a BMC counterexample. We call this automaton a PLC cycle automaton. Afterwards, we gradually add the dynamic behaviour and introduce the parallel composition of two automata as shown in Section 3.2. In the first step in Section 3.3, we extend the PLC cycle automaton by adding a timer and constraints for this timer to model the cycle time of each PLC cycle. This is accomplished by constructing a hybrid automaton which models the cycle time and perform a parallel composition

with the PLC cycle automaton. In following step we add transition guards and assignments for the dynamic behaviour according to a set of rules as presented in Section 3.4. The flows of the plant dynamics are included by constructing another hybrid automaton. This automaton models the dynamics for a list of conditional ODEs. Thereafter, we perform another parallel composition using the previously composition result with the extended guards and assignments and the newly constructed automaton for the conditional ODEs. How the initial assignments for the continuous variables are determined is presented in Section 3.6. The initial values are determined based on conditions over the discrete variables provided by the counterexample. Consequently, we have added all plant dynamics to the model of the counterexample by constructing a hybrid automaton, which is called a counterexample automaton, using these transformations.

As explained in Section 2.6, we use the architecture of the SFC Verification tool, which provides a toolchain with tools to add certain aspects of the dynamic behaviour. This toolchain is presented in Section 3.7, where we explain the tools which are crucial to construct the automaton as described. Furthermore, we give a short overview of some tools which are needed to create suitable models for SpaceEx and Flow*.

in Section 3.8 we propose an extension for the BMC counterexamples. The idea is to allow wildcards instead of fixed value assignments. Subsequently, we show how the construction of the PLC automaton changes if these wildcards occur in the counterexample sequences.

Once the hybrid automaton has been constructed, we can use it to verify whether the counterexample still describes a path where we reach an unsafe state while the dynamic behaviour is taken into account. The result of such an analysis can be used to determine discrete sequences, which cannot occur if the plant dynamics are considered.

3.1 PLC Cycle Automaton

In order to analyze the dynamic behaviour of a plant, we combine the plant dynamics with the counterexamples of the discrete analysis. This allows us to create a reduced hybrid system as we only consider a single discrete path which has been determined to be unsafe. Thus, we check whether this behaviour can occur if the plant dynamics are incorporated. Our goal is to construct an automaton, which extends the counterexample provided by the BMC with the dynamic behaviour, thus allowing us to use hybrid reachability analysis to verify the extended discrete counterexamples.

Firstly, the BMC counterexample is transformed into a hybrid automaton. This PLC cycle automaton represents the sequences of variables given by the counterexample without any plant dynamics. The hybrid automaton as defined in Definition 2.2, which we use to model the PLC cycle automaton, however has the possibility to be extended with the plant dynamics. This allows us to gradually add dynamic behaviour.

The length of the equi-length variable sequences of a counterexample defines the number of automaton locations as each location corresponds to a cycle during the PLC execution. The assignments of the first cycle are provided by the first entries of each sequence of the counterexample and are converted into the initial state of the automaton. Furthermore, the input and output variables are converted into assignments on the transitions between these locations when the sequence is converted into an automaton. This allows us to model the cycle sequence, since we assure that in each cycle the discrete values are assigned according to the given counterexample.

Assuming we have input and output sequences of length n provided by a BMC counterexample as shown in Equations (3.1) to (3.6), we can construct a PLC cycle automaton as presented in Figure 3.2.

$$in_{v_1} : (v_{1,1}, v_{1,2}, \ldots, v_{1,n}) \tag{3.1}$$
$$in_{v_2} : (v_{2,1}, v_{2,2}, \ldots, v_{2,n}) \tag{3.2}$$
$$\vdots \quad : \qquad\qquad \vdots$$
$$in_{v_k} : (v_{k,1}, v_{k,2}, \ldots, v_{k,n}) \tag{3.3}$$
$$out_{v_1} : (v_{k+1,1}, v_{k+1,2}, \ldots, v_{k+1,n}) \tag{3.4}$$
$$out_{v_2} : (v_{k+2,1}, v_{k+2,2}, \ldots, v_{k+2,n}) \tag{3.5}$$
$$\vdots \quad : \qquad\qquad \vdots$$
$$out_{v_m} : (v_{k+m,1}, v_{k+m,2}, \ldots, v_{k+m,n}) \tag{3.6}$$

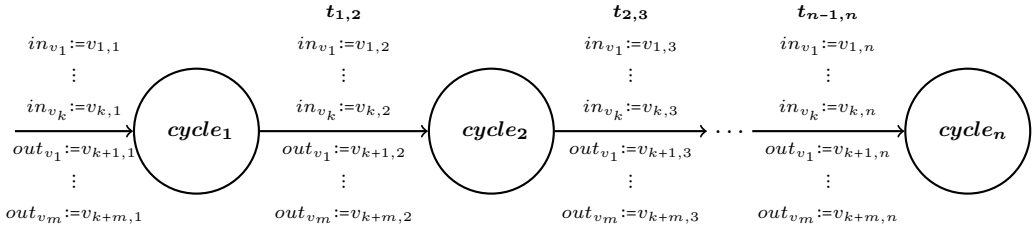

Figure 3.2: *PLC Cycle Automaton*

The discrete behaviour provided by the BMC counterexample is modeled by the PLC cycle automaton. Synchronization labels are assigned to each transition according to the cycle numbers of the source and target locations. Thus, a transition from $cycle_i$ to $cycle_{i+1}$ is labeled with the synchronization label $t_{i,i+1}$. The automaton has no continuous variables and flows, all transition guards and invariants are set to *true*, i.e. all variable valuations are allowed, for each transition and location. We now add the dynamic behaviour to the PLC cycle automaton to construct a counterexample automaton.

3.2 Generate Counterexample Automaton

As our goal is to validate the counterexample, once it has been extended by the dynamic behaviour, the counterexample automaton has to incorporate all plant dynamics. Firstly, we show how we can construct such a counterexample automaton by using the parallel composition on hybrid automata as defined in Definition 2.2 to add plant dynamics.

Using the definition given in Definition 2.2, we are going to generate a hybrid automaton, which contains the plant dynamics. We accomplish this, by constructing several hybrid automata containing different aspects of the dynamic behaviour, for which we construct the parallel compositions. During each parallel composition, we add additional dynamic behaviour.

The parallel composition $\mathcal{H}_1 \| \mathcal{H}_2$ of two hybrid automata $\mathcal{H}_1$ and $\mathcal{H}_2$, results in a hybrid automaton $\mathcal{H}$. In order to construct the parallel composition, the labeling of the transitions has to be considered. Non-synchronizing transitions, i.e., τ-transitions do not synchronize with other transitions and can be taken in the composition automaton independent of the second automatons behaviour. Each hybrid automaton includes a set of τ-transitions, which do not change the state of the automaton as defined in Definition 2.2. All transitions with synchronization labels can only be taken if either the second automaton takes a transition with the same label or if the label set of the second automaton does not include the label and takes a τ-transition.

Furthermore, the possible variable valuations for two hybrid automata $\mathcal{H}_1$ and $\mathcal{H}_2$ are extended. We consider the composition for disjunct sets of variables Var_1 and Var_2. $\forall v \in Var_1 \cap Var_2$, we rename v in one automaton to create disjunct sets. Afterwards, we extend the domain of the valuations for both automata. Assuming V_1 is the set of all valuations $\nu_1 : Var_1 \to \mathbb{R}$ and V_2 is the set of all valuations $\nu_2 : Var_2 \to \mathbb{R}$, the new set of all valuations for both automata is $V = V_1 \cup V_2$. Thus, we can use the new valuation domain to construct the intersection of the valuations in $\mathcal{H}_1$ and $\mathcal{H}_2$, which results in new valuation

for the composition automaton. We define the parallel composition of hybrid automata $\mathcal{H}_1$ and $\mathcal{H}_2$ as shown in Definition 3.1, which results in a new hybrid automaton $\mathcal{H}$.

Definition 3.1 (Parallel Composition HA/HA)
The parallel composition of the hybrid automaton $\mathcal{H}_1 := (Loc_1, Var_1, Lab_1, Trans_1, Flow_1, Inv_1, Init_1)$ *and the hybrid automaton* $\mathcal{H}_2 := (Loc_2, Var_2, Lab_2, Trans_2, Flow_2, Inv_2, Init_2)$ *with extended valuations is a hybrid automaton* $(Loc_p, Var_p, Lab_p, Trans_p, Flow_p, Inv_p, Init_p)$ *where:*

- $Loc_p := Loc_1 \times Loc_2$
- $Var_p := Var_1 \cup Var_2$
- $Lab_p := Lab_1 \cup Lab_2$
- $((l_1, l_2), \lambda, \mu, (l_1', l_2')) \in Trans_p$ *iff*
  - *there exists a transition* $(l_1, \lambda_1, \mu_1, l_1') \in Trans_1$ *and a transition* $(l_2, \lambda_2, \mu_2, l_2') \in Trans_2$ *such that*
  - *either* $\lambda = \lambda_1 = \lambda_2$ *or*

    $\lambda = \lambda_1 \in Lab_1 \backslash Lab_2$ *and* $\lambda_2 = \tau$ *or*

    $\lambda = \lambda_2 \in Lab_2 \backslash Lab_1$ *and* $\lambda_1 = \tau$ *and*
  - $\mu = \mu_1 \cap \mu_2$
- $Flow_p((l_1, l_2)) := Flow_1(l_1) \cap Flow_2(l_2)$ *where* $l_1 \in Loc_1$ *and* $l_2 \in Loc_2$
- $Inv_p((l_1, l_2)) := Inv_1(l_1) \wedge Inv_2(l_2)$ *where* $l_1 \in Loc_1$ *and* $l_2 \in Loc_2$
- $Init_p := \{((l_1, l_2), \nu) | (l_1, \nu_1) \in Init_1 \wedge (l_2, \nu_2) \in Init_2 \wedge \nu = \nu_1 \cap \nu_2\}$

The parallel composition creates new locations Loc_p consisting of all combinations of the locations Loc_1 of $\mathcal{H}_1$ and the locations Loc_2 of $\mathcal{H}_2$. The labels Lab_p of the new automaton are the union of the sets of labels Lab_1 and Lab_2 of the two automata. A transition is added to the new systems if the synchronization labels match. Moreover, we add a transition if $\mathcal{H}_1$ takes a transition with label $a \in Lab_1 \backslash Lab_2$ and $\mathcal{H}_2$ takes a τ-transition as well as for the analogous case where $\mathcal{H}_2$ takes a transition with a label which does not occur in Lab_1. The guards and transitions are composed by intersecting the transition relations μ_1

and μ_2, thus creating the intersection of the valuations defining the guards and assignments. The new flows $Flow_p$ for a location (l_1, l_2) are created by adding the intersection of the flows given by $Flow_1$ and $Flow_2$ for locations l_1 and l_2 respectively. This way the composition models the flows by adding the flows of l_1 and l_2 to the combined location (l_1, l_2). The invariants Inv_1 and Inv_2 for the locations l_1 and l_2 are added in conjunction for the combined location (l_1, l_2) in Inv_p a both invariants have to be satisfied for the composition automaton to be able to reach location (l_1, l_2). The new initial states $Init_p$ are constructed from the intersection of the original initial states $Init_1$ and $Init_2$ as only the states need to be considered, where both automata are in an initial state.

We use the parallel composition to extend the PLC cycle automaton which is constructed as shown in Section 3.1.

3.3 PLC Cycle Times

A PLC executes its programming cyclically and requires a certain amount of time for each cycle as explained in Section 2.1. These cycle times are part of the PLCs dynamic behaviour and have to be added to model the dynamic behaviour correctly. We can define cycle times for the PLC in three different ways. The cycle time can be defined as

1. a constant time c for each PLC cycle or
2. an upper and lower bound $[l, u]$ for each PLC cycle or
3. an upper and lower bound $[l_i, u_i]$ for each PLC cycle i individually.

We now show the construction of a hybrid automaton for the third case, as the other two cases can be simply derived from this construction. Assuming we have the cycle times $[l_i, u_i]$ for each PLC cycle i of the counterexample, we construct an automaton of the same length, i.e., the same amount of locations since we need a location for each cycle similar to the given PLC cycle automaton derived from the counterexample as presented in Section 3.1. The hybrid automaton for the parallel composition is illustrated in Figure 3.3.

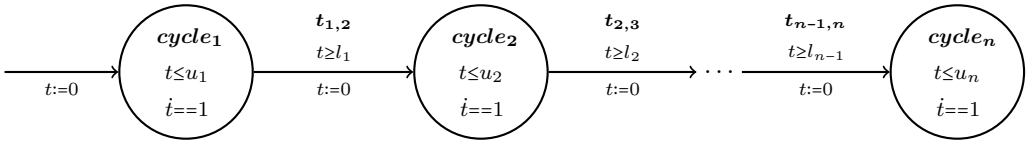

Figure 3.3: *PLC Cycle Times Automaton*

The new automaton has a similar sequential structure as the PLC cycle automaton. Each location of the hybrid automaton contains a flow for the time variable t which defines a constant passing of time. Furthermore, in each cycle location i we require an invariant $t \leq u_i$ and the guards to its successor are $t \geq l_i$ and a reset assignment $t := 0$. The combination of these restrictions and the assignments allows us to model the upper and lower bounds for each PLC cycle as the system can only stay between l_i and u_i in each cycle i. The transitions are labeled in the same way as the PLC cycle automaton presented in Figure 3.2.

We now apply the parallel composition $\mathcal{H}_1 \| \mathcal{H}_2$ for automaton $\mathcal{H}_1$ as defined Figure 3.2 and automaton $\mathcal{H}_2$ as defined in Figure 3.3. If we omit all unreachable locations and transitions in $\mathcal{H}_1 \| \mathcal{H}_2$, the composition results in the automaton shown in Figure 3.4, where we rename all composition locations $(cycle_i, cycle_i)$ to $cycle_i$ in the new automaton.

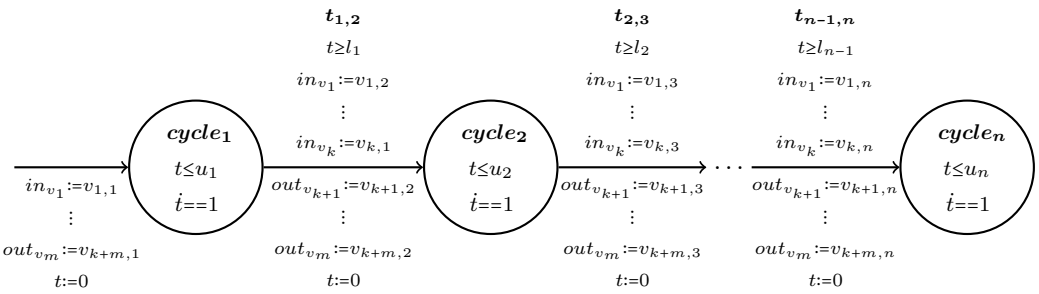

Figure 3.4: *Composition Automaton*

The resulting automaton combines the discrete behaviour of the BMC counterexample as well as the dynamic behaviour given by the PLC cycle times.

3.4 Discrete/Dynamic Linking

The discrete assignments given by the counterexamples also provide information about the plant dynamics. The linking between the discrete and dynamic behaviour is given by replacement rules as presented in Section 3.4.1. We use these replacement rules to model the dynamic behaviour of the systems sensors by defining restrictions of the continuous variables. We show how to apply these replacement rules to a hybrid automaton in Section 3.4.2.

3.4.1 Replacement Rules

The replacement rules define the relations between the variables provided in the BMC counterexample and the continuous variables of the conditional ODEs. These relations can be used to connect the discrete sensors and variables to the dynamic behavior of the system and are stored in *Link Files*. Currently, the replacement rules support boolean and integer values. There are two different types of *Link File* entries.

Firstly, an error $\epsilon > 0$ is specified, which allows us to model $<$ and $>$ for verification tools which are restricted to $\leq$ and $\geq$. Thus, using small values for ϵ to model strict inequalities for the replacement rules, we get the transformations seen in Equations (3.7) and (3.8).

$$l < r \implies l \leq r - \epsilon \tag{3.7}$$

$$l > r \implies l \geq r + \epsilon \tag{3.8}$$

The second type of entries connects the discrete with the dynamic behaviour. We define a replacement rule as shown in Definition 3.2.

Definition 3.2 (Replacement Rule)
A replacement rule $\rho := cond \Leftrightarrow guard(\Leftrightarrow assign)$ *describes the relation between the discrete and dynamic behaviour of a system for a set discrete variables* $Vars_d$ *and a set of continuous variables* $Vars_c$*. Furthermore,* $Conds_d$ *is the set of all possible conditions for* $Vars_d$ *and* $Conds_c$ *is the set of all possible conditions for* $Vars_c$*.*

- $cond \in Conds_d$ *defines a condition over* War_d*. The rule is only applied if cond is satisfied.*
- $guard \in Conds_c$ *describes the new guard for the transition over* Var_c *(optional if* true*)*
- *assign is a set of variable valuations* $\nu : Vars_c \to \mathbb{R}$ *for the continuous variables defining new assignments for a transition*

A replacement rule ρ defines a relation between discrete variables and continuous variables. Each of these entries consist of two or three sides which are separated by $\Leftrightarrow$. The left side of the entry *cond* defines a condition over the discrete variables. If during the automaton generation the assignments of the discrete variables in a cycle satisfy the condition *cond*, the other sides *guard* and *assign*

of the entry are conjuncted with the current transition guard and provide additional transition assignments. In this case, we say the replacement rule is satisfied. All satisfiable replacement rules are applied in conjunction as new transition guards and assignments. The replacement rules allow overlapping conditions as well as conditions, which do not cover the entire domain of the variables used in the conditions.

For the assignment of each transition, we check which replacement rules are applied to the transition. The satisfiability of each condition is checked using SMTInterpol which has been presented in Section 2.7. In order to use the SMT solver, we construct a formula representing the condition of the replacment rule and the discrete variable assignments. For each condition we assign a set of equalities and inequalities to define the variable assignments according to the sequence. Thus, an assignment of a single value c to a variable v results in a equation $v = c$, which added in conjunction with the actual condition *cond*. Furthermore, we can also model the assignment of intervals $[l, u]$ to v, by adding two inequalities $v >= l$ and $v <= u$ where l and u are numerical values.

3.4.2 Transition Dynamics

The guards and assignments for the continuous variables are determined according to the discrete variable assignments using the replacement rules introduced in Section 3.4.1. We present such a transformation with the aid of an exemplary set of replacement rules and a counterexample. The replacement rules are applied to all transitions in the transformed automaton shown in Figure 3.4.

A possible entry for a boolean variable in_v and continuous variable d is the replacement rule `in_v == 1 <=> d > 10`. If the negated condition is required to model the plant dynamics as well, an entry `in_v == 0 <=> d <= 10` has to be added to the set of replacement rules. An exemplary set of replacement rules for the discrete input variables in_{v_1}, in_{v_2} and in_{v_3} and the continuous variables d_1 and d_2 is shown in Equations (3.9) to (3.11).

$$in_{v_1} == 1 \Leftrightarrow d_1 \geq 20 \Leftrightarrow d_2 := 0 \tag{3.9}$$

$$in_{v_2} == 0 \wedge in_{v_3} < 10 \Leftrightarrow d_2 \leq 50 \wedge d_1 < 30 \tag{3.10}$$

$$in_{v_3} >= 10 \Leftrightarrow d_1 < 10 \tag{3.11}$$

Assuming we apply the replacement rules to an exemplary transition representing cycle i in a hybrid automaton with guard g and the assignments $in_{v_1} := 1$, $in_{v_2} := 0$ and $in_{v_3} := 5$ omitting all other assignments. We get the transformation shown in Figure 3.5 for a transition of an extended PLC cycle automaton.

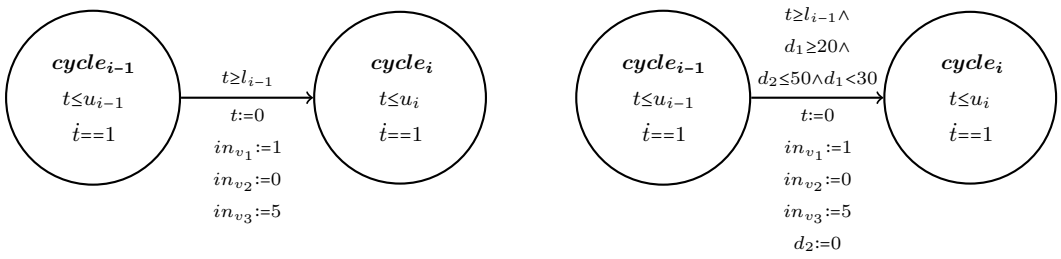

Figure 3.5: *Replacement Rule Application*

For a transition in the PLC Cycle automaton which has been extended with the cycle times, Figure 3.5 shows the original transition on the left side and the transformed transitions where the replacement rules have been applied on the right side. As the conditions of the replacement rules Equations (3.9) and (3.10) are satisfied, their guards $d_1 \geq 20$, $d_2 \leq 50 \wedge d_1 < 30$ and assignment $d_2 := 0$ are added in conjunction to the current transition guard g and the given transition assignments respectively.

This transformation is applied to all the transitions of the automaton given in Figure 3.4. Then the guards have been adapted to incorporate restrictions for the dynamic behavior, but the flows and their conditions have not yet been added. We define these conditions and the related dynamic behavior by conditional ordinary differential equations as described in Section 2.6.1.

3.5 Adding Conditional ODEs

The conditional ODE systems as introduced in Section 2.6.1 are added to the hybrid automaton presented in Figure 3.4 after the replacement rules have been applied as described in Section 3.4.2. This transformation is used to add the flows for the continuous variables to each location as well as restrict the flows according the conditions of the conditional ODE systems.

We now construct a hybrid automaton which represents the conditional ODE systems. By adding this automaton, we want to model the behaviour of the continuous variables for each condition and the ODEs. The general idea of the composition is to create an automaton representing the dynamic behaviour defined by conditional ODE systems, thus by composing it with the extended PLC cycle automaton, i.e. the PLC cycle automaton which has been extended with the cycle times and where the replacement rules have been applied, we extend each location which represents a PLC cycle with dynamic behaviour of the conditional ODE systems.

As the condition of a conditional ODE system defines when the ODEs are applied as flows we add the condition to the invariant of a location and the ODEs as flows. We model all conditional ODE systems by adding one location for each conditional ODE system. If the conditions of the conditional ODE systems do not cover all possible valuations, we need to construct a negation for the conditions by creating the conjunction of all negated conditional ODE system conditions, i.e. $cond_{neg} = \neg cond_1 \wedge \neg cond_2 \wedge \ldots \wedge \neg cond_l$, and add an additional location, where the flows $ODEs_{neg}$ represent chaotic behaviour for all continuous variables defined in the ODEs of the conditional ODEs. Furthermore, if the invariant would no longer be satisfied if a time step is taken, we need transitions to the locations representing the other conditional ODEs allowing the system to switch to another location where the invariant is satisfied by the current state. The conditions must overlap for such a location change to be able to occur. We call these connecting transitions *copy transitions*. All locations are defined as initial locations, because this allows the system to start its analysis in any conditional ODE system where the condition is satisfied. The automaton representing the conditional ODEs shown in Equations (2.12) to (2.14) is illustrated in Figure 3.6.

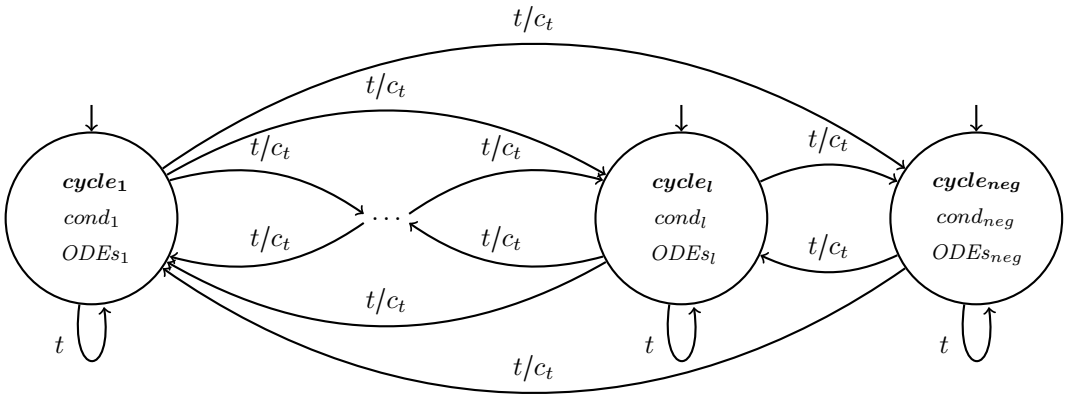

Figure 3.6: *Conditional ODE Automaton*

Each transition with the label t/c_t defines two transitons, where one transition is labeled with t and the second transition is labeled with c_t. The transitions labeled c_t are the *copy transitions*, while the transitions labeled t are added so that the parallel composition adds all possible transitions from a location corresponding to cycle i to all locations corresponding to cycle $i+1$ for $i \in \{1, 2, \ldots, n-1\}$.

The hybrid automata in Figure 3.4 after replacement rules as described in Section 3.4.2 have been applied is adapted by replacing all labels $t_{i,i+1}$ with t for

$i \in \{1, 2, \ldots, n-1\}$. We now compute the parallel composition of this automaton and the automaton in Figure 3.6. Each location in Figure 3.4 is extended by the behaviour of the newly constructed automaton. The intersection of synchronization label sets is not empty since t is used in both automata. However as c_t only occurs in the conditional ODE system automaton, thus these transitions are only synchronized with τ-transitions. The resulting parallel composition is shown in Figure 3.7 where the location combination $(cycle_i, cycle_j)$ is renamed to $cycle_{i,j}$, where $cycle_i$ is the location of the extended PLC cycle automaton and $cycle_j$ is the location in the conditional ODE automaton.

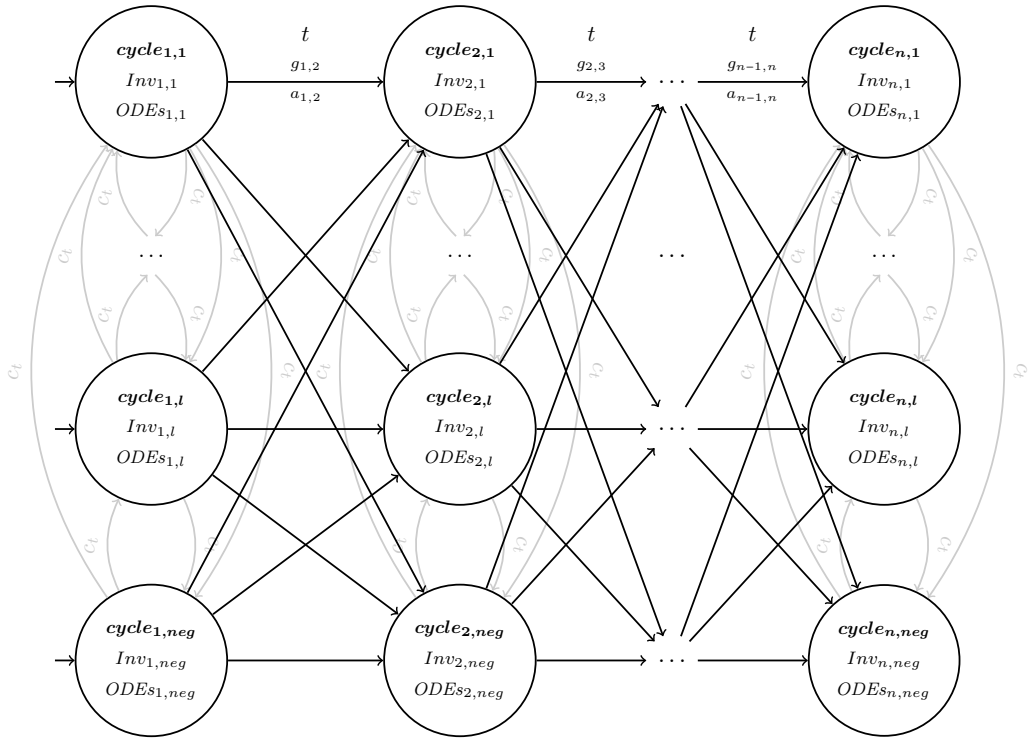

Figure 3.7: *Parallel Composition*

All transitions connecting a location $cycle_{i,u}$ representing cycle i with a location $cycle_{i+1,v}$ describing the successor cycle $i+1$ where $u, v \in \{1, \ldots, l, neg\}$ are assigned the label t, guard $g_{i,i+1}$ and assignments $a_{i,i+1}$. The guards $g_{i,i+1}$ correspond to the guards given by the corresponding transitions in Figure 3.4 in conjunction with the guards provided by the application of the replacement rules. Furthermore, the transitions of this automaton determine the assignments $a_{i,i+1}$. The new invariants $Inv_{i,v}$ where $v \in \{1, \ldots, l, neg\}$ for a location of cycle i consist of the conditions of the conditional ODE in conjunction with the time constraint $t \leq u_i$. As the only type of flows in the automaton is the time

flow $\dot{t} == 1$ each set of ODEs $ODEs_{i,v}$ in cycle i is the set of flows provided by $ODEs_v \cup \{\dot{t} == 1\}$ for $v \in \{1, \ldots, l, neg\}$. The assignments of the initial states correspond to the initial assignments of the first cycle in Figure 3.4 as the second automaton provides no new assignments.

The resulting automaton now models the discrete behaviour given by the BMC counterexample as well as the dynamic behaviour defined by the replacement rules and conditional ODE systems. The actual transformation is performed by a toolchain as introduced in Section 2.6.2 containing tools to transform hybrid automata in several steps.

3.6 Conditional Initial Values

The initial values for the continuous variables have not yet been defined. We cannot simply set constant values for the continuous variables of the system, as their values may depend on the activities of discrete variables. Thus, we introduce conditional initial values, where each initial value of a continuous variable depends on a condition.

Conditional initial values define conditions for each continuous variable of the hybrid system for which specific values are assigned to the variable. A definition for conditional initial values is given in Definition 3.3.

Definition 3.3 (Conditional Initial Value)
A conditional initial values $cond : assign$ *of variable* c *where* $Vars_d$ *is a set of discrete variables and* $Conds_d$ *is the set of all possible conditions over variables* $Vars_d$

- $cond \in Conds_d$ *is the condition over the discrete variables* $Vars_d$
- $assign \subset \mathbb{R}$ *is a set of valuations for* c.

If the condition *cond* of a conditional initial values is satisfied in a hybrid system for the given initial discrete variable assignments, the continuous variable d will be assigned according to *assign*. The conditional initial values support the assignment of constant values as well as the assignment of intervals for numerical values or the assignment of sets. The conditions of the set of conditional initial values do not have to be complete and are allowed to overlap. If two conditional initial values are satisfied for the same initial discrete assignments due to an overlap of their conditions, both initial assignments are added to the initial

states. An exemplary set of conditional initial values for the water level h of the tank system presented in Section 2.2 is given in Equations (3.12) to (3.16).

$$in_{\text{full}} : \; h := 25 \tag{3.12}$$
$$\neg in_{\text{full}} \wedge in_{\text{max}} : \; h := 17 \tag{3.13}$$
$$\neg in_{\text{max}} \wedge in_{\text{min}} : \; h := 10 \tag{3.14}$$
$$\neg in_{\text{min}} \wedge in_{\text{nonempty}} : \; h := 4 \tag{3.15}$$
$$\neg in_{\text{nonempty}} : \; h := -1 \tag{3.16}$$

The given conditional initial values for h describe the sections of the tank, which are defined by the discrete sensors. There are five different sections since there are 4 water level sensors. These initial values should be chosen according the sensor placements in the plant. The conditional initial values are assigned to an automaton using a new tool which is added to the toolchain which has been introduced in Section 2.6.2. During the analysis we also consider the set of initial values, where for each area all possible values are assigned. Restricting the values of the water level to $h \in [-30, 50]$ we can define the conditional initial values in Equations (3.17) to (3.21), which correspond to the water level sensors.

$$in_{\text{full}} : \; h := [20, 50] \tag{3.17}$$
$$\neg in_{\text{full}} \wedge in_{\text{max}} : \; h := [15, 20) \tag{3.18}$$
$$\neg in_{\text{max}} \wedge in_{\text{min}} : \; h := [5, 15) \tag{3.19}$$
$$\neg in_{\text{min}} \wedge in_{\text{nonempty}} : \; h := [0, 5) \tag{3.20}$$
$$\neg in_{\text{nonempty}} : \; h := [-30, 0) \tag{3.21}$$

3.7 Automaton Toolchain

The remaining transformations for the automaton are performed by a toolchain, which allows us to transform automata into a verifiable model. The toolchain has been developed for the SFC Verification tool, which allows the user to verify sequential function charts. The SFC Verification tool utilizes several toolchain tools to add the flow of continuous variables to automaton in multiple steps and to transform the hybrid automaton into an automaton with a syntax which can either be parsed by SpaceEx or Flow* by eliminating > and <, and disjunction in transition guards and location invariants.

Firstly, we apply tools to add the flows for the continuous variables as introduced in Section 3.7.1 and Section 3.7.2 as shown in Section 3.5. Conditional ordinary

differential equations as presented in Section 2.6.1 are used to define the flows of the continuous variables. Furthermore, we use a tool to create a single initial location in Section 3.7.3, which also models the conditional initial values as presented in Section 3.6. We perform this transformation as the initial values of the continuous variables depend on the discrete behaviour and Flow* does not support multiple initial locations. In order to support interval and set assignments, we also provide a tool to resolve these assignments in Section 3.7.4. A short description of tools which allow us to transform an automaton into a SpaceEx or Flow* verifiable model is given in Section 3.7.5.

3.7.1 Adding Dynamic Behavior

The *Add Dynamic Behaviour* tool preforms one part of the transformation presented in Section 3.5. Assuming the same input, the tool splits each location of the input automaton into $l+1$ locations for a list of conditional ODEs of size l. The $(l+1)$th location describing the location with the negated conditional ODE conditions. Furthermore, the transitions from a cycle to its successor are added as explained in Section 3.5. The automaton resulting from the transformation of the tool, given a PLC cycle automaton is illustrated in Figure 3.8.

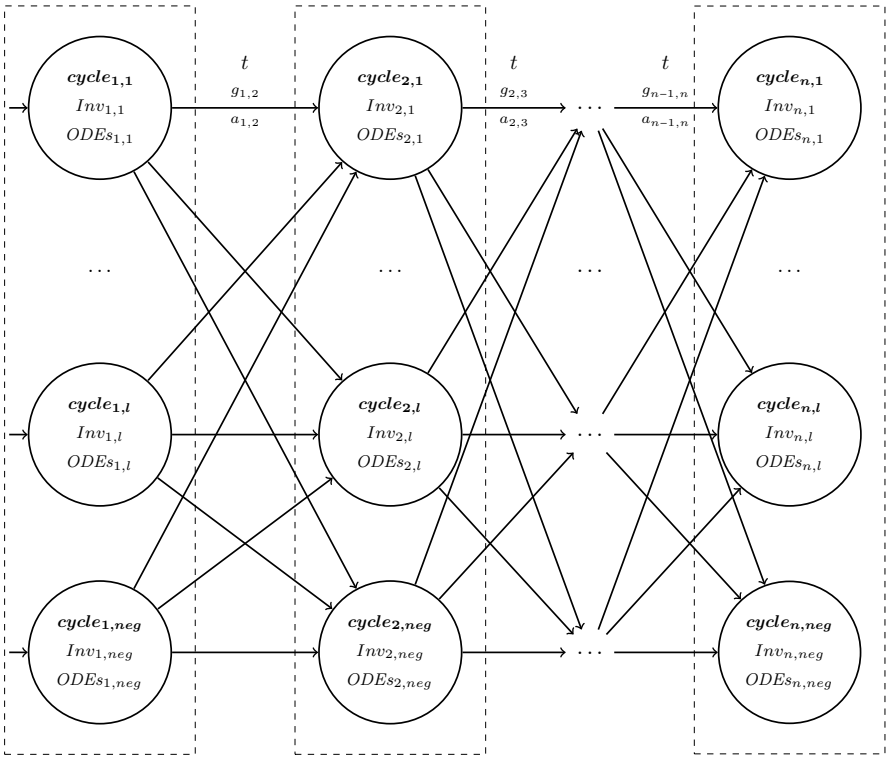

Figure 3.8: *Add Dynamic Behaviour Transformation*

The dashed rectangles contain sets of cycle locations, which are logically grouped, as they represent the same location in the original counterexample, thus representing the same PLC cycle. These groupings are maintained in order to restore the path taken in the original automaton by examining the path taken in the transformed automaton.

We can omit the *copy transitions* as presented in Section 3.5 if the conditions of the conditional ODEs only contain restrictions on the discrete variables as seen for the tank system in Equations (2.15) and (2.16) since the discrete values cannot change during the execution of a cycle location. For all other cases, we add the *copy transitions* using another tool from the toolchain.

3.7.2 Copy Transitions

As described before, an invariant representing the condition of a conditional ODE system may become unsatisfiable, but another one is still satisfiable, the system must be able to change between the corresponding locations. These changes are only allowed if both invariants are satisfied, thus they implicitly provide guards for the *copy transitions*. For our case, the *copy transitions* are only needed if the conditions of the ODEs contain constraints on the continuous variables as the discrete behaviour can only change if a new cycle is entered.

In addition to the previously described tools, we use a tool to create transitions between grouped locations to allow the system to switch between locations which correspond to the same location in the original automaton. These transitions do not have a guard or any assignments.

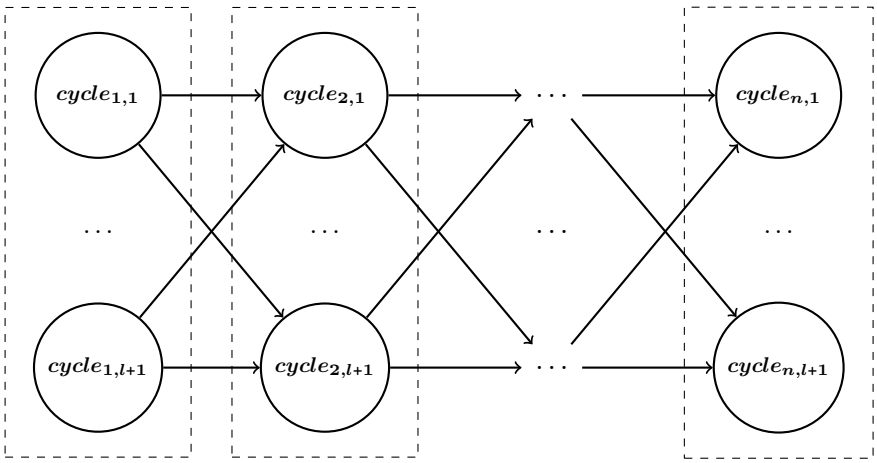

Figure 3.9: *Automaton before copy transitions have been added*

The locations framed by the dashed rectangles seen in Figure 3.9 are locations, that have been grouped together by the automaton construction or an automaton tool. For a group $C_i := \{cycle_{i,1}, \ldots, cycle_{i,l+1}\}$ we add a new *copy transition* for each location $cycle_{i,j}$ with $j \in \{1, \ldots, l+1\}$ to each location $cycle_{i,k}$ where $k \in \{1, \ldots, l+1\} \backslash \{j\}$ with $i \in \{1, \ldots, n\}$.

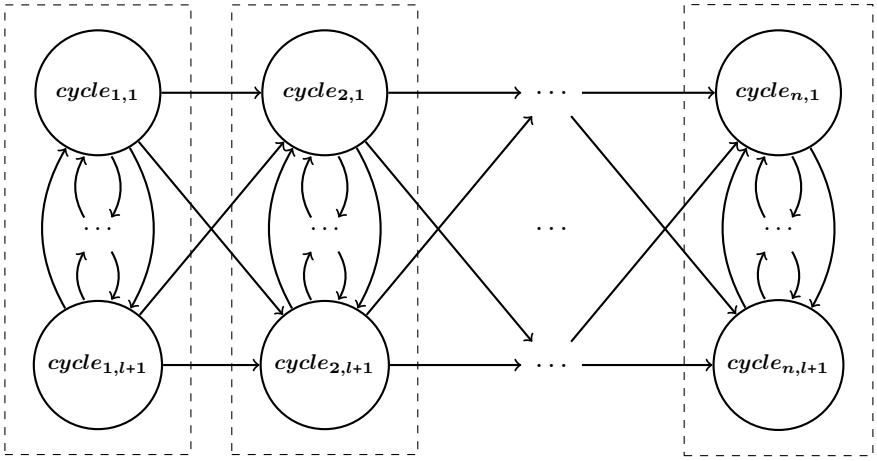

Figure 3.10: *Automaton after copy transitions have been added*

Figure 3.10 shows the automaton after the execution of the tool where *copy transitions* have been added between all cycles in each group.

3.7.3 Single Initial Location

Since some tools do not allow multiple initial locations, we create a new location loc_{init} which incorporates all other initial locations by adding transitions from loc_{init} to all original initial locations.

The previously presented construction of a labeled hybrid automaton as explained in Section 3.2 and Section 3.5 results in several initial states, where each initial state has the same variable assignments. This is the case, as the initial state is derived from the first assignments of each discrete variable of the BMC counterexample is copied to all locations corresponding to the first PLC cycle. These initial assignments are transferred to the new initial location loc_{init}. In order to force the system out of the new initial location loc_{init} immediately, we define the flow for the time $t := 0$ as $\dot{t} == 1$ and an invariant $t \leq 0$. This combination of this dynamic behaviour prevents the system from staying in the initial location. All the flows for all other continuous variables are set to 0 as they should not change in the initial location.

The initial values only contain the initial values of the discrete variables derived from the BMC counterexample and for $t := 0$. The initial values of the continuous variables need to be linked to the these initial values. Thus, we use conditional initial values as described in Section 3.6, which consist of conditions and related initial assignments. For each conditional initial value we add a new transition from loc_{init} to each original initial locations using the condition as a transition guard and the initial value as an assignment. Thus, the initial value is assigned if the guard is satisfied by the initial discrete assignments. If there are multiple continuous variables, their conditional initial values are used to produce all combinations of conditions and possible assignments. The continuous variables can be initialized with 0 in the new initial state, as the actual initialization of the continuous variables is accomplished by the constructed transitions. An exemplary set of conditional initial values of a set and an interval is illustrated in Equations (3.22) and (3.23).

$$in_{sen} : \ h := \{0, 1, 2, 3\} \tag{3.22}$$

$$\neg in_{sen} \wedge out_{val} > 10 : \ h := [4, 20] \tag{3.23}$$

We add two transitions from the new initial location to each original initial locations. The guards of each pair corresponds to the conditions of the initial values, thus they are in_{sen} and $\neg in_{sen} \wedge out_{val} > 10$ respectively. The automaton Figure 3.11 shows a model with multiple initial states with different locations with a set of initial assignments $init$ for the discrete variables. Furthermore, as the initial assignments are accomplished by the transformation we use the assignment of 0 for the actual initial states.

After the transformation, a new location loc_{init} has been added and only one initial state remains. The conditions of the conditional initial values are used as guards for the new transitions from loc_{init} to the original initial locations $loc_1, loc_2, \ldots, loc_m$. Furthermore, $ODEs_0$ represents the set where for all continuous variables d their ODE is $\dot{d} == 0$ except for $\dot{t} == 1$. The resulting automaton is illustrated in Figure 3.12.

In order to make sure that only one condition is satisfiable, during the construction of the automaton we add the negated predecessor conditions in conjunction with the transition guard. In this example, we can omit this negation, as we would construct the conjunction of $\neg in_{sen}$ and $\neg in_{sen} \wedge out_{val} > 10$.

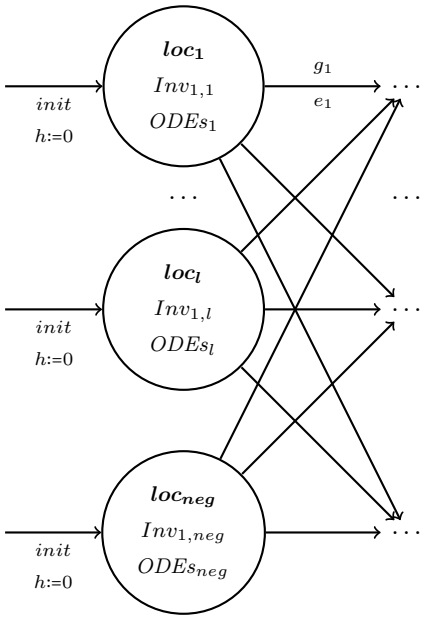

Figure 3.11: *Exemplary Set of Initial States*

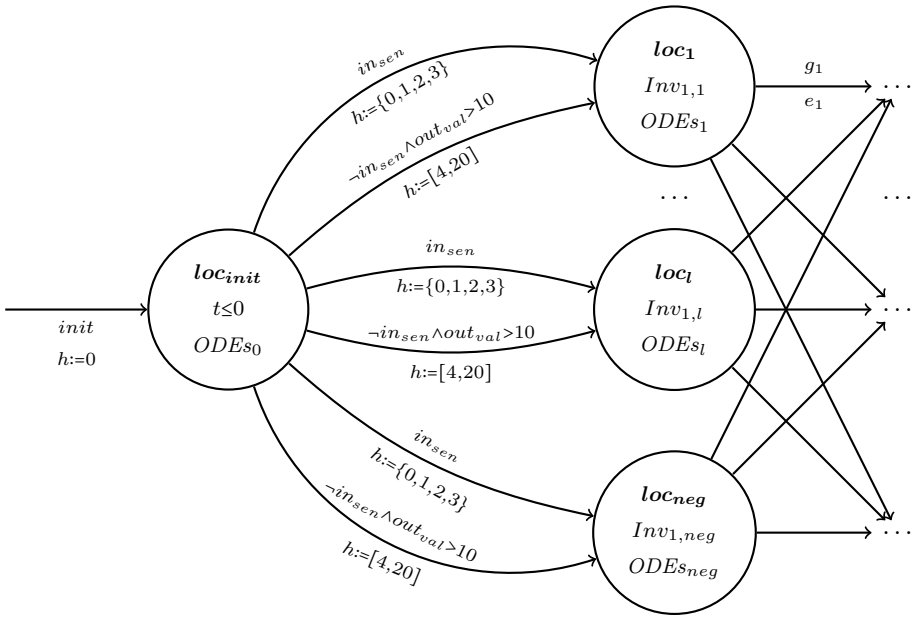

Figure 3.12: *Transformed Set of Initial States*

As we already employ SMTInterpol as presented in Section 2.7 to check the satisfiability of replacement rules, a possible improvement would be to create different inital states according to the satisfiability of the conditions of the conditional initial values given the discrete initial values derived from the BMC counterexample. This method is only applicable, if the verification tool supports multiple initial locations.

3.7.4 Interval and Set Assignments

Transition assignments for numerical variables can either be a single value, an interval or a set of values. In order to verify these assignments we transform the interval and set assignments into single value assignments. We introduce a tool which converts interval and set assignments.

In the first step, all interval assignments are transformed. The idea for the interval transformation is to create a new location, where the system is allowed to assign the variable, which has an interval assigned, all possible values in the interval. For each interval assignment $v \coloneqq [l_v, u_v]$ on a transition from loc_1 to loc_2 a new location loc_v is generated. Furthermore loc_v has the invariant $v \leq u_v$. Thus, we ensure, that if the location may only be left if the upper bound of the variable is still satisfied. All differential equations $Var \backslash v$ for all variables $Var \backslash \{v\}$ are set to 0 and the differential equation for v is set to 1. The transition original transition from loc_1 to loc_2 is replaced by two new transitions from loc_1 to loc_v and loc_v to loc_2, where the transition from loc_1 to loc_v has the same guard γ as the original transition and the assignment $v \coloneqq l$. This in combination with the flows prevents v from assuming values below the lower bound, the assignment is only performed if the guard is satisfied and all other variables stay the same. Furthermore, all remaining assignments α are transferred to the transition from loc_v to loc_2. This transition does not have a guard. Therefore, after the interval assignment all other assignments are applied. The full transformation of a single interval assignment is illustrated in Figures 3.13 and 3.14.

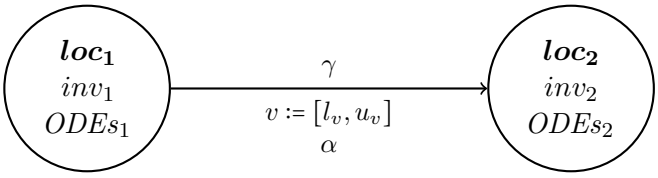

Figure 3.13: *Interval Assignment*

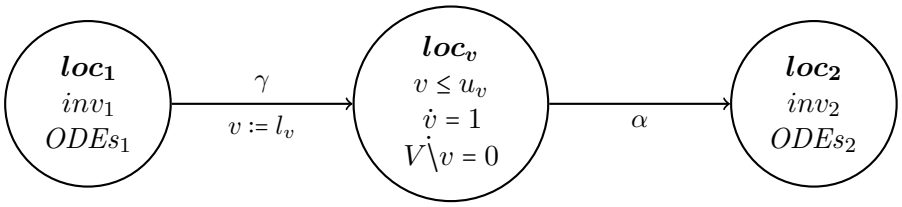

Figure 3.14: *Transformed Assignment*

This transformation allows v to be assigned any value between l_v and u_v. The variable v is initially assigned value l_v, when the automaton enters loc_v. The system can stay in loc_v at most for $u_v - l_v$ time units and is able to leave loc_v at any time before. Thus, v can have values between l_v and u_v when entering loc_2. If A still contains interval assignments, the transformation is repeated for remaining interval assignments.

Set assignments are transformed by the same tool. They are transformed after interval assignments, as they create multiple transitions, which would multiply interval assignments as well. The idea is to create a new transition for each element from the set and assign a different value of the set to the variable on each transition. A set assignment $v := \{a_1, a_2, \ldots, a_n\}$ of transition from location loc_v to loc_2 is transformed into n new transitions. These new transitions also have source loc_1 and target loc_2. Each of these transitions assigns a different value from $\{a_1, a_2, \ldots, a_n\}$ to v. The guard γ and remaining assignments α are added to each of these transitions. Figure 3.15 shows the transition before the transformation and Figure 3.16 illustrates the newly constructed transitions.

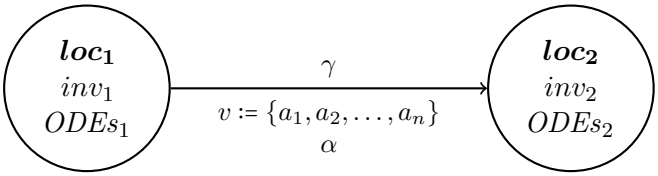

Figure 3.15: *Set Assignment*

The transformation allows assignment of any value from the set $v := \{a_1, a_2, \ldots, a_n\}$ by choosing the corresponding transition. If α still contains set assignments, the transformation is repeated for all remaining set assignments. Both transformations can be accomplished in linear time.

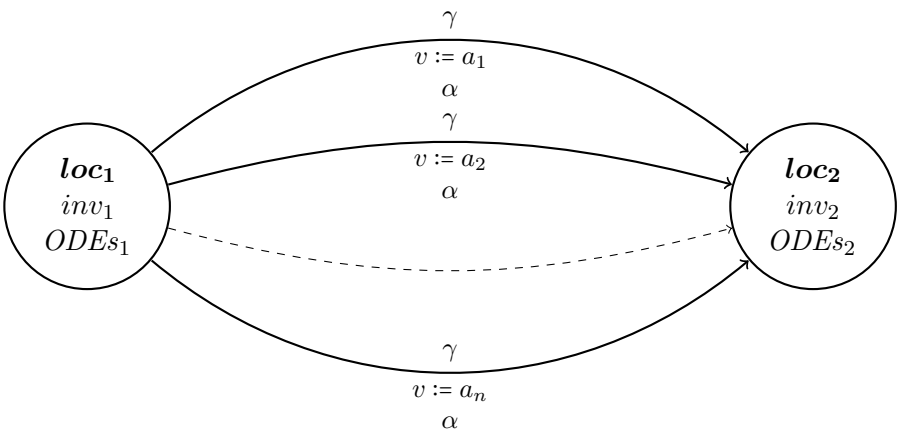

Figure 3.16: *Transformed Assignment*

3.7.5 Additional Tools

In addition to the previously described tools, we employ additional tools to apply further transformations to the automaton. These tools allow us to construct hybrid automata, which have a syntax supported by either SpaceEx or Flow*.

As disjunction is not supported for some verification tools, we need to transform any transition guard, which includes disjunctions. Initially, the transition guard is transformed into disjunctive normal form (DNF). Afterwards, transitions are split for each clause and the clauses are the new guards of the split transitions. A second tool uses similar approach to split disjunction in invariants of locations.

In order to reduce the size of the automaton, we also remove all locations with unsatisfiable invariants as well as all incoming, outgoing transitions and initial states which include the location. Such locations occur for example, if the conditions of all conditional ODEs are complete. In this case the negated condition of all conditional ODEs is unsatisfiable.

After these tools have been applied, we have constructed the counterexample automaton containing all plant dynamics. Furthermore, the resulting labeled hybrid automaton has been transformed into an automaton with a syntax which is supported by SpaceEx or Flow*, depending on the applied tools. For example, in order for the model to be verifiable by SpaceEx, disjunctions have to be resolved as well as strict inequalities. On the other hand, Flow* requires for all initial states to share the same initial location as well.

3.8 Wildcard Values

The presented counterexamples can be extended to incorporate wildcards. A wildcard represents the assignment of all possible values for a given data type of a variable to the variable. Wildcards are defined in Definition 3.4.

Definition 3.4 (Wildcard)
A wildcard $\omega(v) = S$ is an assignment of all possible values for a variable v according to its data type.

- *Boolean has the possible values $S = \{true, false\}$*
- *Integer has the possible values $S = \mathbb{Z}$*

Using wildcards in counterexamples, their expressiveness can be increased significantly. We are able to represent partial cycle sequences, which are sufficient to represent a counterexample. An example of a sequence containing wildcards for a reduced tank system as presented in Section 2.2 given in Listing 3.1.

```
(in_max:bool,in_min:bool,out_v:bool)
(*,*,*,0,0,1,0,0,0)
(*,*,*,0,1,1,*,*,*)
(*,*,*,*,*,*,*,*,*)
```

Listing 3.1: *Counterexample with Wildcards*

The given counterexample states, that no matter how the discrete variables are set in the first three cycles, the BMC still detected a counterexample. The fourth to sixth cycles however, have to exhibit the discrete behaviour for `in_max` and `in_min` as described in the counterexample. Furthermore, in the last three cycles `in_max` has to be assigned 0 in each cycle.

Currently, we restrict our usage of wildcards to boolean values. Each wildcard for a boolean value results in two cases for the current PLC cycle as we can either assign the value to *true* or *false*. A wildcard for an integer value would $\mathbb{Z}$ however results in a countable but infinite set if possible values. This poses a problem when we try to construct the possibilities for an integer value. This feature will be implemented in future work.

3.8.1 Counterexample Wildcards

The counterexamples presented in Section 2.3.3, can be extended to incorporate wildcards as described in Section 3.8. These wildcards have to be handled differently during the assignment of transition guards and assignments than a constant assignment of a value. There are two different cases for the automaton transformation, if a wildcard occurs in a sequence. For now, only boolean wildcards are supported in the automaton construction.

Firstly, when one or more wildcards appear as the first values in a sequence, the initial states have to be adapted. The new initial states have to contain all combinations of possible wildcard values. Assuming we have an arbitrary number of non-wildcard initial assignments *assign*, these would create exactly one set of initial assignments. For each initial boolean wildcard, the amount of initial states is doubled, as there are exactly two possible assignments 0 and 1 representing $false$ and $true$ for a boolean variable.

For each initial state s and a new boolean wildcard $\omega(v)$ for variable v, we create remove s and construct two new initial states $s_{v:=0}$ and $s_{v:=1}$. Initial state $s_{v:=0}$ contains the assignment of $s := 0$, while $s_{v:=1}$ includes the assignment $s := 0$. Figure 3.17 illustrates the transformation of a initial state into two new initial states due to a boolean wildcard.

Figure 3.17: *Initial Wildcard Assignment*

The left initial state shows the state s before transformation, while the right automaton shows the initial states $s_{v:=0}$ and $s_{v:=1}$. This transformation is repeated for all initial states and all wildcards. Thus, for all initial states and a new wildcard assignment, we apply the presented transformation on all initial states. The resulting set of new initial states is then used as a starting point for additional new wildcard variables and their transformations. As mentioned before, each new boolean wildcard doubles the number of initial states, due to the transformation being applied to all initial states.

The second transformation, which must be adapted for wildcards, is the transformation of the transition guards and assignments. As explained in Section 3.4.1, we use the given assignments for a cycle to determine, which replacement rules

are satisfied. If wildcards occur in the assignments of the a cycle, we must check all possible combinations of assignments for the wildcards. In the initial construction presented in Section 3.2 and Section 3.4.2, we create a transition from each cycle $cycle_i$ location to its successor $cycle_{i+1}$. We now add a transition between $cycle_i$ and $cycle_{i+1}$ for each assignment combination. We then assign guards and assignments according to the replacement rules.

Assuming we have an assignment for cycle $cycle_{i+1}$, with three boolean wildcard variables in_{v_1}, in_{v_2} and out_{v_3} as well as an arbitrary number of non-wildcard assignments $assign$. Furthermore, we assume the replacement rules given in Equations (3.24) and (3.25).

$$in_{v_1} == 1 \wedge in_{v_2} == 1 \Leftrightarrow cond_1 \Leftrightarrow assign_{cont} \tag{3.24}$$

$$in_{v_1} == 1 \vee in_{v_2} == 1 \Leftrightarrow cond_2 \tag{3.25}$$

For better comprehension, we divide the transformation into two parts. In the first part we separate the transitions for all possible input variable combinations. We further split these transitions in the second part, where we assign all possible output variable values to the different transitions. As we have two boolean wildcard input variables, we obtain four transitions. The transitions for $in_{v_1} = *$, $in_{v_2} = *$ and the non-wildcard assignments $assign$ for the discrete variables and the given replacement rules are illustrated in Figure 3.18.

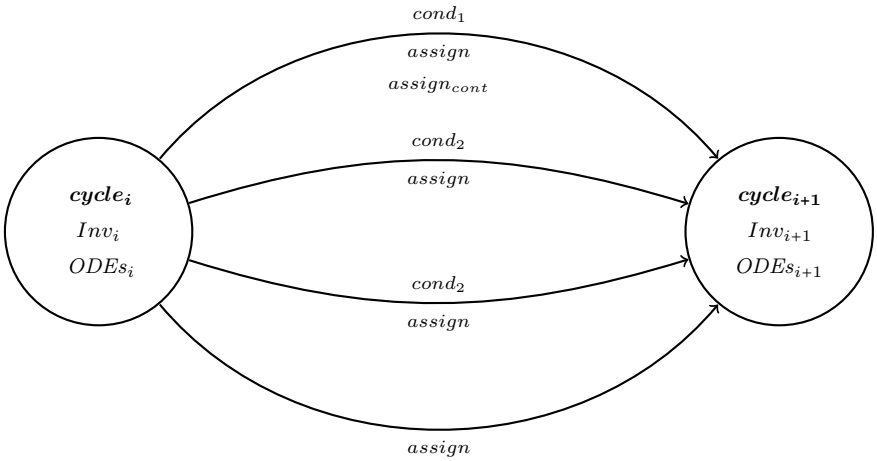

Figure 3.18: *Transition Input Wildcard Transformation*

Starting from the top transitions, the transitions represent the discrete input variable assignments $in_{v_1} = 1$ and $in_{v_2} = 1$,$in_{v_1} = 1$ and $in_{v_2} = 0$,$in_{v_1} = 0$ and

$in_{v_2} = 1$ and $in_{v_1} = 0$ and $in_{v_2} = 0$. In addition to the input variables, we need to consider the output wildcard variables. In this example, the only output variable, which is a wildcard is out_{v_3}. Each of the transitions created in the previous step are now split for each output wildcard. We show the exemplary transformation of the $in_{v_1} = 1$ and $in_{v_2} = 0$ transition in Figure 3.19.

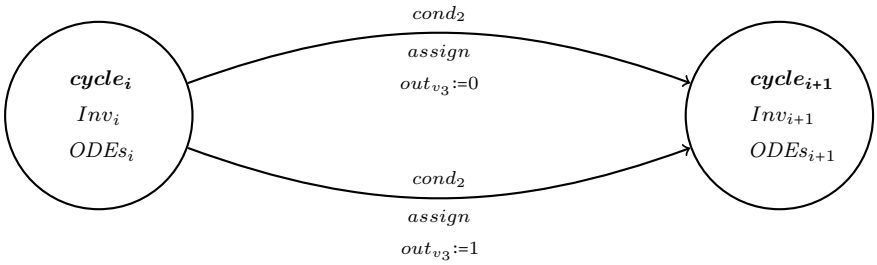

Figure 3.19: *Transition Output Wildcard Transformation*

The second transformation can be achieved by assigning the values of the boolean output variables as set assignments. For example in this case, we can simply assign each transition with $out_{v_3} := \{0, 1\}$. Afterwards, the transformation proposed in Section 3.7.4 for set assignments can be used.

After both transformations have been applied, we have 2^3 transitions between $cycle_i$ and $cycle_{i+1}$. The presented transformation allow us to model wildcards in a counterexample. We can model wildcards in the initial values as well as wildcards in the other cycle assignments.

3.9 Summary

In this chapter we have presented the transformation of a BMC counterexample into a hybrid automaton which contains all plant dynamics. The transformation is performed gradually. In the first step, we construct an automaton representing the discrete behaviour described by the counterexample. This results in a sequential automaton, where each location corresponds to a PLC cycle. Afterwards, we show how to create an automaton modeling the cycle times for each PLC cycle and use the parallel composition on these two automata. The transition guards and assignments for continuous variables of the resulting automaton are extended using replacement rules. Thereafter, we introduce an automaton representing the dynamic behaviour provided by conditional ODE systems. This automaton is used to construct another parallel composition with the automaton that resulted from the last transformation. The last step which adds plant dynamics consists of initializations of the continuous variables.

These initializations depend on the initial assignments of the discrete variables provided by the counterexample. We present the tool chain which is part of the architecture of the SFC Verification tool. The tools allow us to add the plants dynamic behaviour to hybrid automata and to transform these automata in order to create a suitable model for SpaceEx and Flow*. Furthermore, we introduce wildcards and propose transformations to handle wildcard assignments in counterexamples for discrete boolean values.

Chapter 4

Hybrid Counterexample Analysis

In this chapter the different input formats for the plant dynamics are discussed. We show how we can use XML files to represent conditional ODEs in a way that we can repurpose the parsers of the SFC Verification tool as presented in Section 2.6. Furthermore, we extend these XML files to incorporate the conditional initial values. A format to store the replacement rules as presented in Section 3.4.1 is also introduced. As the hybrid reachability is computed using third-party tools, we discuss how to configure these tools according to the structure of the generated automaton. The PLC cycle times provide information about the settings referring to the time horizon and time step sizes. Moreover, we address how the amount of computational steps of the reachability analysis can be restricted.

The XML format for the conditional ODEs and conditional initial values is shown in Section 4.1. Additionally, we show how to store the replacement rules and how to define the PLC cycle times. Afterwards, we discuss the parameters, which are used by SpaceEx and Flow* in Section 4.2. The time and iteration parameters are set according to the generated automaton.

4.1 Input Parameters

In this section, we define the input parameters for the tank example as defined in Section 2.2. Firstly, we define the plant dynamics using conditional ODEs and conditional initial values and show how a XML file can be used to store these dynamics in Section 4.1.1. The replacement rules as presented in Section 3.4.1 can be stored as shown in Section 4.1.2. In Section 4.1.3 we show how to define the PLC cycle times for hybrid automaton as well as how to parametrize the tool for our novel approach.

4.1.1 Dynamic Behavior

In order to add the dynamic behavior missing from the original model, we define conditional ODE systems and conditional initial values as described in Section 2.6.1 and Section 3.7.3.

We define the conditional ODE systems in a XML file, where each conditional ODE system consists of a single condition and at least one ODE. The following XML file as shown in Listing 4.1 models the in Equations (2.15) and (2.16) previously determined conditional ODE system.

```
<?xml version="1.0" encoding="UTF-8"?>
<condODEsys>
    <condODE>
        <cond><![CDATA[out_v]]></cond>
        <equation>h' == 2</equation>
    </condODE>
  <condODE>
        <cond><![CDATA[NOT out_v]]></cond>
        <equation>h' == -2</equation>
    </condODE>
    ...
</condODEsys>
```

Listing 4.1: *Conditional ODE System*

The modeled behaviour describes the two possible states of the valve of the tank system introduced in Section 2.2. The first conditional ODE system models the incoming flow of water 2 for h under the condition, that the valve is open, thus out_v. The second conditional ODE system shows represents the water level change of -2 if the valve is closed, i.e. $\neg out_v$.

The tag `<condODEsys>` contains all conditional ODEs, which are defined in `<condODE>` tags each. The `<condODE>` tags define a conditional ODE system and contains at least two entries. The first entry `<cond>` contains the condition of the conditional ODE system, which uses a custom syntax for logic formulas. This syntax supports logic operators and comparators to check integer and real values. The list of supported operators and comparators is shown in Table 4.1.

	Operator	**Description**
NOT	$\neg$	Negation of a boolean value
AND	$\wedge$	Logic conjunction
OR	$\vee$	Logic disjunction
==	=	Equality comparator
>,<,>=,<=	$>, <, \geq, \leq$	Inequality comparators

Table 4.1: *Custom Condition Syntax*

All other entries inside `<condODE>` define the ODEs, where each `<condODE>` contains at least one ODE. Each ODE is stored in `<equation>` tags. In this example, the ODEs depend on the state of out_v. If the valve is open, the water

height inside the tank increases by 2 units per time unit. On the other hand, if the valve is closed the water height decreases by 2 per time unit.

The presented format for the conditional ODEs is used in the SFC Verification tool which has been presented in Section 2.6. We extend the XML file, so it can define conditional initial values in addition to the conditional ODE systems.

For each continuous variable a set of conditions and associated initial values is assigned. The XML file for the exemplary set of conditional initial values as defined in Equations (3.12) to (3.16) for the discrete variables $in_{\text{full}}, in_{\text{max}}, in_{\text{min}}, in_{\text{nonempty}}$ and continuous variable h is shown in Listing 4.2.

```
<?xml version="1.0" encoding="UTF-8"?>
<condODEsys>
    ...
    <init>
      <variable var = "h">
        <condInit>
          <cond><![CDATA[in_full]]></cond>
          <value>25</value>
        </condInit>
        <condInit>
          <cond><![CDATA[in_max AND NOT in_full]]></cond>
          <value>17</value>
        </condInit>
        <condInit>
          <cond><![CDATA[NOT in_max AND in_min]]></cond>
          <value>10</value>
        </condInit>
        <condInit>
          <cond><![CDATA[NOT in_min AND in_nonempty]]></cond>
          <value>4</value>
        </condInit>
        <condInit>
          <cond><![CDATA[NOT in_nonempty]]></cond>
          <value>-1</value>
        </condInit>
      </variable>
    </init>
</condODEsys>
```

Listing 4.2: *Conditional Initial Values*

The conditional initial values for each variable are defined in the `<init>` tags. For each continuous variable v, we create a new set of tags `<variable>` with an attribute `var` which contains the variable name v. Furthermore, each conditional initial values is contained in a pair of tags `<condInit>` similar to a conditional ODE system. This tag contains exactly two entries. The `<cond>` tag defines a condition *cond* using the same custom syntax (Table 4.1) the conditional ODE systems use. The `<value>` tags are used to set the value which is assigned to v if the condition *cond* is satisfied. As explained in Section 3.6, these values

can be constants, intervals or sets of values. A set $\{v_1, \ldots, v_n\}$ by setting value between the `<value>` tags to `{v_1,...,v_n}`. For an interval $[l, u]$ assignment the tag value is set to `[l,u]`.

In order to define open intervals as shown in Equations (3.17) to (3.21), the values have to be defined with an error, similar to the approximation shown in Section 3.4.1. Thus, $[5, 15)$ has to be set to `[5,14.99999]`.

The conditions of the tank example are disjunct presented in Listing 4.2 as we have defined $in_{\text{full}} \Rightarrow in_{\text{max}} \wedge in_{\text{min}} \wedge in_{\text{nonempty}}$, $in_{\text{max}} \Rightarrow in_{\text{min}} \wedge in_{\text{nonempty}}$ and $in_{\text{min}} \Rightarrow in_{\text{nonempty}}$. In this case, each conditional initial value for h represents a different water height section of the tank.

4.1.2 Link File

The conditional ODEs and conditional initial values are not sufficient to describe all plant dynamics. We need to define the links between the discrete and dynamic behaviour as described in Section 3.4. The *Link File* stores the replacement rules as described in Section 3.4.1. The first line defines the error *epsilon* which is used to approximate < and >. All consequent lines define a replacement rule for a specific input sensor. An exemplary link file is shown in Listing 4.3.

```
epsilon = 0.00001
in_full == 1 <=> h >= 20
in_max == 1 <=> h >= 15
in_min == 1 <=> h >= 5
in_nonempty == 1 <=> h >= 0
in_full == 0 <=> h < 20
in_max == 0 <=> h < 15
in_min == 0 <=> h < 5
in_nonempty == 0 <=> h < 0
```

Listing 4.3: *Link File*

The first line `epsilon = 0.00001` defines error $\epsilon := 0.00001$. The following lines link the values of input variables with the continuous variable. As described in Section 3.4.1, each entry corresponds to a possible guard on each transition. Thus, to model the sensors correctly, two replacement rules have to be defined. The first representing a sensor that is active, for example `in_full == 1 <=> h >= 20` and a second, where the sensor is not active `in_full == 0 <=> h < 20`. The same principle applies to all other entries, as these variables are boolean as well. The link table in Figure 4.1 shows all obtainable links of discrete to continuous variables for this special case, where the *Link File* also contains the negation for each condition of a replacement rule.

Variable	*true*	*false*
in_{full}	$h \geq 20$	$h < 20$
in_{max}	$h \geq 15$	$h < 15$
in_{min}	$h \geq 5$	$h < 5$
in_{nonempty}	$h \geq 0$	$h < 0$

Figure 4.1: *Variable links*

These links are used to model the guards of transitions according to the sequence as described in Section 3.4.1.

4.1.3 Properties

In Addition to the conditional ODEs and the *Link File*, path parameters and cycle times are defined in the property file for the hybrid counterexample analysis. The property file shown in Listing 4.4 is used for a verification using the SpaceEx tool platform.

```
SEQUENCE_FILE_LOCATION=/home/.../bmcCounterexample.txt
COND_ODE_FILE_LOCATION=/home/.../condODEs.xml
LINK_FILE_LOCATION=/home/.../linkDynamics.txt
RESULT_FILE_LOCATION=/home/.../result.xml
SPACEEX_FILE_LOCATION=/home/.../spaceExConfig_phav.cfg
FLOWSTAR_FILE_LOCATION=/home/.../flowStarConfig.cfg
CYCLE_TIMES=1
```

Listing 4.4: *Property File*

The parameter `SEQUENCE_FILE_LOCATION` specifies, where the counterexample analysis tool is expecting the counterexample, which is provided by the BMC analysis. `COND_ODE_FILE_LOCATION` defines the location of the XML file containing the conditional ODEs and conditional initial values. The *Link File* is specified in `LINK_FILE_LOCATION`. The file path set for `RESULT_FILE_LOCATION` describes where the model of the hybrid system is placed. SpaceEx expects a XML file, while Flow* needs a *.model file to verify the system. Additionally, `SPACEEX_FILE_LOCATION` and `FLOWSTAR_FILE_LOCATION` provide the paths to the verification tools configuration files.

The last parameter `CYCLE_TIMES` is used to set the cycle times of each PLC cycle. As shown in Section 3.3, there are three different possiblities to define these times. We can either set a constant c, the same interval $[l, u]$ for each cycle or an individual interval for each cycle. The last definition can be used by providing a list of intervals $[l_1, u_1], \ldots, [l_n, u_n]$ where each interval corresponds to the minimal and maximal time of a PLC cycle.

The bounded model checker as presented in Section 2.3 uses these property files to interface with the hybrid analysis and define the PLC cycle times.

4.2 Reachability Analysis

In order to perform a meaningful analysis, certain parameters of the tools which we utilize for hybrid reachability analysis have to be set according to the behaviour of the PLC. Furthermore, we have to consider the length of the counterexample and total number of locations.

In this section we discuss some of the parameters for the tools used for the hybrid reachability analysis, i.e. SpaceEx and Flow*. The restrictions on the cycle times as presented in Section 3.3 can be used to determine the settings the verification tools. We show how to determine bounds for these parameters in Section 4.2.1. In Section 4.2.2 we show, how we can determine bounds for the parameters pertaining the the computation of timed and discrete steps.

4.2.1 Time Parameters

In this section we discuss how the structure of the automaton and the PLC behaviour affect the time parameters of the reachability analysis. The hybrid verification tools we employ use a time horizon to restrict the time elapse computations. We differentiate between a local and global time horizon. If a local time horizon t is specified, during the computation of all reachable successor states of a state, the time elapses are bounded by t. A global time horizon T restricts the computation of all reachable states to the states reachable to the time interval $[0, T]$ in a hybrid automaton.

Firstly, we define the time constraints on the analysis according to the PLC cycle times which have been presented in Section 3.3. The cycle times provide us with an upper bound on the global time horizon as utilized in Flow* of the analysis as they can be used to compute the sum of all maximal PLC times. This sum can be used as the upper bound as it represents the maximal time the PLC needs to execute all cycles and due to the time invariants added during the transformation in Section 3.3. The upper bound can be easily computed for all three cases of the PLC cycle times, i.e. individual cycle time intervals for each cycle, a single cycle time interval for all locations and a constant time for all cycles. We now consider the most general case, where we have individual intervals for each cycle. Equation (4.1) show how to compute the upper bound t_g for the global time horizon of the hybrid analysis, assuming we have n PLC cycles with cycle time $[l_i, u_i]$ for each cycle i.

$$t_g := \sum_{i=1}^{n} u_i \tag{4.1}$$

If we assume the same cycle time interval $[l, u]$ for each PLC cycle, the formula is adapted with $u_i = u$ for $i \in \{1, \ldots, n\}$. Similarly, for a constant c cycle time we get $u_i = c$ for $i \in \{1, \ldots, n\}$. If the verification tool allows to set a local time horizon as in SpaceEx, we set the upper bound t_l for the local time horizon as defined in Equation (4.2) for individual PLC cycle time intervals.

$$t_l := \max_{i \in \{1,\ldots,n\}} \{u_i\} \tag{4.2}$$

The largest upper limit of all cycle times is an upper bound as the analysis may not stay longer than t_l in any cycle location due to the invariants added during the transformation described in Section 3.3. If the cycle times are restricted by the same interval $[l, u]$ for each cycle, the time horizon is bounded by $t_l = u$. The upper bound $t_l = c$ can be used for the constant time c.

Furthermore, the presented upper bounds can be used to bound the number of time steps, which will be performed in each cycle. For example, we can ensure that the analysis performs at most s steps by setting the time step size of the verification tool to t_g/s. A step size to ensure that at least s steps are performed can be computed by considering the sum over all lower bounds of the PLC cycle times instead of t_g. This is assuming the system cannot change between more than s conditional ODE location without Zeno behaviour occuring in the cycle.

If interval assignments are used in the automaton, the bounds have to be adapted, since the locations resulting from a transformation of an interval assignment as shown in Section 3.7.4, might increase the upper bounds t_l and t_g. Another solution is to adapt the flow of the variable of the interval assignment in this interval assignment location in order to avoid the computation of a large time horizons in these locations.

4.2.2 Iteration Parameters

Considering the structure of the counterexample automaton presented in Chapter 3 and the bounds on the maximal number of steps performed in each PLC cycle as shown in Section 4.2.1. We discuss how we can bound the maximum jump depth, i.e. the maximal number of discrete steps, and the number of iterations, i.e. the number of time elapses of the local time horizon and successor computations. In Figure 4.2 we illustrate a reachability tree to explain the difference of jump depth and number of iterations.

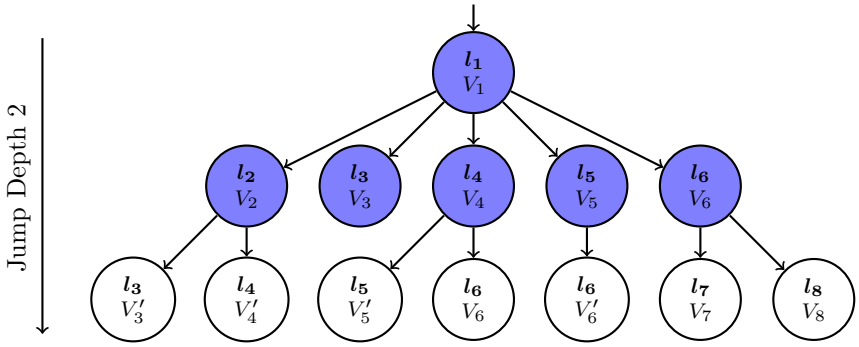

Figure 4.2: *Exemplary Reachability Tree*

The tree in Figure 4.2 shows the reachable states of an exemplary hybrid automaton which is restricted by a jump depth or a number of iterations. Assuming all reachable states have been computed for the first three levels in the tree, the tree could be the result of an analysis with maximum jump depth 2 as it contains all states which are reachable after at most 2 discrete steps. However if we assume the analysis was restricted by a specific number of iterations, the blue states represent the states for which all reachable states for a given time horizon t have been computed. Thus the analysis was restricted by 6 iterations. The jump depth defines how many levels in the reachability tree are computed, while the iterations define how many states are analyzed by computing the reachable set of states, which are reachable from this state.

Flow* allows the user to define a maximum jump depth. The construction of the counterexample automaton for n cycles, creates a single initial location with invariants, which force the analysis to leave the location immediately. Thus, after 1 discrete step, the analysis is in a location corresponding to the first PLC cycle. We might need to consider locations which are constructed during the interval assignment transformation as presented in Section 3.7.4 as well. Assuming we can perform at most s time and discrete steps in a PLC cycle, due to the restrictions provided by the fixed step size and location invariants as described in Section 4.2.1. As the system can perform at most s time elapses in each cycle, the analysis can perform at most s discrete steps. If Zeno behaviour occurs in a cycle, it might be possible to take infinite amounts of jumps in the cycle. However, if the next cycle is reachable, it is still reached after analyzing s levels in the reachability tree. Additionally changing to the next PLC cycle is 1 discrete step bounding the discrete steps for a PLC cycle change at $(s+1)$. In order to reach the last PLC cycle n, these discrete steps have to be executed $n-1$ times as we have already entered the first cycle. The jump depth $1+(s+1)(n-1)$ ensures that if a cycle location of the last PLC cycle is reachable, it will be found during the analysis while restricting computation of all reachable states by the given jump depth.

Unfortunately, SpaceEx does not allow the specification of a maximum jump depth. We can however restrict the number of iterations during the reachability analysis as described in Section 2.5.1. SpaceEx performs a breadth-first search while computing the reachable set of states. Assuming an automaton for n cycles and m locations for all possible conditional ODE systems, we can compute the maximum amount of iterations that are needed considering the specific structure of a model. A SpaceEx iteration computes all reachable states, i.e. all time and jump successors, for a state given a local time horizon t. In the tree constructed during the reachability analysis each level defines the jump successors states of the states of the previous level reachable within t while the first level defines the initial states.

Assuming during the hybrid analysis, the system can switch between different locations belonging to the same cycle. The required number of iterations, which is needed to reach the last cycle, can be computed by considering the number of conditional ODEs m including the negation and PLC cycles n. Assuming the worst case, where all m ordinary differential equation locations are reachable from the initial state, we have to perform $(m+1)$ iterations to compute a jump depth of 2 for the system. One iteration is computed for the initial state and m iterations for all reachable conditional ODE locations. On each subsequent level we assume that for each location all other conditional ODE locations $m-1$ are reachable by using *copy transitions* and all conditional ODE locations m in the next PLC cycle are reachable. Thus the number of states resulting in the next level is $(2m-1) * m$. On each following tree level the successors can be approximated considering the number of possible reachable conditional ODE system locations.

If there is no possibility of Zeno behaviour due to the restriction on the time and sequential structure of the automaton, for example if the system can not switch between conditional ODE locations as in the tank system, we do not need to restrict the number of iterations as at most the n levels in the reachability tree are computed. However, if the system is able to switch between conditional ODE system locations, Zeno behaviour can occur and defining no iteration limit is no longer sensible as the reachability analysis might perform many iterations before terminating. In some models it may only be meaningful to switch the between conditional ODE locations once. For example if we consider two conditional ODE systems describing a gate g being closed, i.e. its value being lowered to 0, and staying closed at 0 as shown in Equations (4.3) and (4.4).

$$g \geq 0 : \dot{g} == -45 \tag{4.3}$$

$$g \leq 0 : \dot{g} == 0 \tag{4.4}$$

The conditions have to be intersecting in order for the automaton to be able to switch between the location corresponding to the same cycle and the given conditional ODE systems. The switch in this case can occur at $g = 0$ in a PLC cycle. The problem is, once the gate is fully closed, the system can switch between the locations after each time elapse as the intersection of the two conditional ODE systems is always satisfied. The system however should stay in the location corresponding to the conditional ODE system (Equation (4.4)) where the gate stays closed.

During each conditional ODE system location switch, a new level is added to the tree, before the automaton changes into the successor PLC cycle location. Thus, if we have information about the amount of conditional ODE system location changes, we can restrict the number of iterations by computing the amount of states on each level for the amount of levels required. Thus we are able to appropriately restrict the number of iterations.

4.3 Summary

In this chapter we show how to store the plant dynamics required for the automaton transformation. We introduced the XML format of the SFC Verification tool, which is used to specify the plant dynamics. This XML file allows us to model the conditional ODE systems for our model. Furthermore, we propose an extension of this file for the conditional initial values. This extension allows us to define conditions and associated initial values for each continuous variable separately. Thereafter, a technique to store the replacement rules is presented. An interface is introduced to allow the BMC to define the PLC cycle times. The hybrid reachability analysis is configured according to the generated hybrid automaton. We show how the PLC cycle times affect the time horizon of the analysis and how the step size for the time steps can be configured. Moreover, we discuss how the cycle times and the structure of the automaton affect the number of iterations of the reachability analysis.

Chapter 5

Explanation Generation

This chapter deals with the generation of explanations for the BMC verification. Explanations provide sets of sequences representing discrete paths, which can be excluded during the BMC counterexample generation as their respective hybrid automata cannot be used to replicate these counterexamples considering the plant dynamics. In order to determine such sequences, the paths visited during the hybrid reachability analysis have to be analyzed. The goal is to either determine if the counterexample has been confirmed given the dynamic behaviour or to compute discrete counterexamples which can be excluded from the BMC analysis in the next iteration, as they are not a valid counterexample given the dynamic behaviour.

In this section, we introduce explanations for the BMC analysis generated by the hybrid analysis. Furthermore, methods to derive the reachable paths from outputs of the previously presented hybrid verification tools are presented. The reachability analysis gives us information about which counterexamples cannot occur if the dynamic behaviour is added or it confirms the counterexample given by the BMC. If the counterexample is confirmed, the result is *unknown* as the confirmation might be the result of over-approximation during hybrid reachability analysis. These information are analyzed by considering the reachable paths and stored as explanations. The explanations are used to exclude certain counterexamples during the next iteration of the BMC. In addition, we discuss how the BMC processes an explanation. Moreover, wildcards, which allow free variable assignments in the sequences provided by an explanation, are introduced to improve the expressiveness of explanations, because a sequence containing wildcards defines multiple discrete paths. We show how wildcards enhance the explanations as well as what restrictions still remain. The functionality described in the chapter is illustrated in Figure 5.1.

A definition for explanations is provided in Section 5.1. Section 5.2 shows how to derive the reachable paths for the SpaceEx and Flow* and store them in a suitable data structure. Using these path, we construct explanations as shown in Section 5.3. The bounded model checker uses these explanations to improve its analysis as explained in Section 5.4. In Section 5.5 we improve the expressiveness of explanations by extending the assignments by wildcards similar to the approach for counterexamples shown in Section 3.8.1.

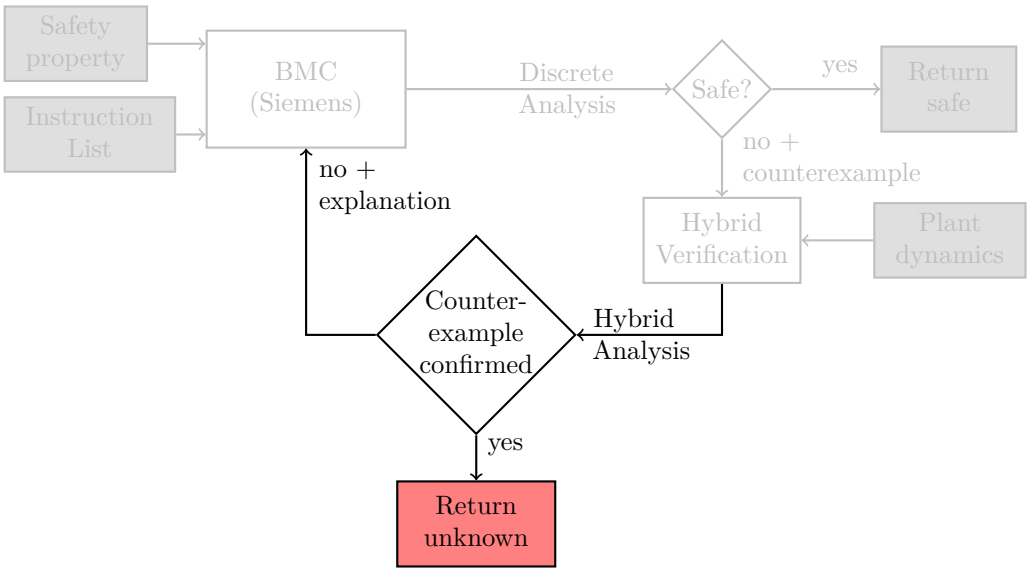

Figure 5.1: *Explanation Generation*

5.1 Explanations

The BMC as presented in Section 2.3 can use sequences of variable assignments for the discrete variables to exclude counterexamples from subsequent analysis iterations. We define explanations to model these sequences. A counterexample will be excluded if the explanation is a prefix of the counterexample. A definition for an explanation is given in Definition 5.1

Definition 5.1 (Explanation)
An explanation $(Vars, Seqs)$ *describes a prefix of a counterexample* $(Vars_c, Seqs_c)$ *where*

- $Vars := Vars_c = \{var_1, \ldots, var_m\}$ *is a set of variables*
- $Seqs := \{s_{1,k}, s_{2,k}, \ldots, s_{n,k}\}$ *is a set of equi-length sequences* $s_{i,k} := (v_{i,1}, v_{i,2}, \ldots, v_{i,k})$ *of assignments for* $var_i \in Vars$ *with* $Seqs_c = \{s_1, s_2, \ldots, s_n\}$, $s_i = (v_{i,1}, v_{i,2}, \ldots, v_{i,n})$ *and* $i \in \{1, 2, \ldots, m\}$ *with* $k \leq n$.

The explanation $(\varnothing, \varnothing)$ *does not exclude any counterexamples and is passed to the BMC if a counterexample has been confirmed. The explanation* $(Vars, \varnothing)$ *excludes all counterexamples.*

An explanation for the bounded model checker is similar to the counterexamples generated by the BMC. The counterexamples as shown in Section 2.3.3 consist

of sequences of variable assignments for the discrete variables of the BMC. An explanation describes a prefix of these sequences. Thus, an exemplary explanation $(Vars, Seqs)$ of a counterexample $(Vars_c, Seqs_c)$ with length n and for all variables $var_i \in Vars$ and variable assignments $v_{i,j}$ for $i \in \{1, 2, \ldots, m\}$ and $j \in \{1, 2, \ldots, k\}$ is given in Equations (5.1) to (5.3).

$$var_1 : (v_{1,1}, v_{1,2}, \ldots, v_{1,k}) \tag{5.1}$$
$$var_2 : (v_{2,1}, v_{2,2}, \ldots, v_{2,k}) \tag{5.2}$$
$$\vdots \quad : \qquad \vdots$$
$$var_m : (v_{m,1}, v_{m,2}, \ldots, v_{m,k}) \tag{5.3}$$

These explanations are stored in a file using the same format as the counterexamples provided by the bounded model checker. These files can be used by the BMC to improve the analysis as described in Section 2.3.2. An exemplary explanation for the tank example presented in Section 2.2 is shown in Listing 5.1.

```
(in_full:bool,in_max:bool,in_min:bool,in_nonempty:bool,out_v:bool)
(0,0,0,0,0,0,0,0,0)
(0,0,0,1,1,1,0,0,0)
(1,1,1,1,1,1,1,1,0)
(1,1,1,1,1,1,1,1,1)
(1,1,1,1,0,0,0,0,0)
```

Listing 5.1: *Tank System Explanation*

In order to obtain explanations for a given counterexample, we have to consider the reachable paths of the hybrid reachability analysis on the transformed automaton. The reachable paths can be derived from the outputs given by SpaceEx and Flow*.

5.2 Reachable Paths

In this chapter we discuss, how to obtain reachable paths from different verification tool outputs. The reachable paths can be used to determine, which discrete counterexamples are reproducible in the hybrid automaton. Section 5.2.1 describes how the console output provided by a modified SpaceEx can be used to construct a reachability tree as shown in Section 2.5.3. Furthermore, Section 5.2.2 shows how to configure Flow* to generate the path information needed to derive a complete reachability tree.

5.2.1 SpaceEx Reachability Tree

The SpaceEx output allows us to (re)construct a path tree for the analyzed model. Thus, the tree contains all used transitions and visited locations. During this tree construction, states that have already been visited do not appear in the tree. The information for the reachability tree is obtained from the extended console output of SpaceEx for the verbosity D2, which defines an output with debug level 2. The original SpaceEx output does not allow to detect, when a state has to be added to the waiting list, which is a queue of states that still needs to be analyzed by SpaceEx. An excerpt of an exemplary console output for a automaton `cycleSequence`, which is derived from a BMC counterexample is shown in Listing 5.2.

```
...
00:00:01.015 1 sym states passed, 1 waiting
00:00:01.015 Iteration 0 done after 0.253001s
00:00:01.015 Iteration 1...
00:00:01.015
time elapse in loc(cycleSequence)==cycle1_1
00:00:01.015 Continuous post with continuous_post_sfm...
00:00:01.036 max error 0 in direction -in_reset
00:00:01.041 Continuous post with continuous_post_sfm done after 0.0250001s,
    cumul 0.0530001s
00:00:01.041
discrete post with label "" from loc(cycleSequence)==cycle1_1 to loc(
    cycleSequence)==cycle2_6
source location: loc(cycleSequence)==cycle1_1
00:00:01.042
discrete post with label "" from loc(cycleSequence)==cycle1_1 to loc(
    cycleSequence)==cycle2_5
source location: loc(cycleSequence)==cycle1_1
00:00:01.042 Discrete post...
00:00:01.044 computing discrete post of sfm...  of size 11
00:00:01.044 found 1 intervals intersecting with guard
00:00:01.060 computing discrete post of sfm done after 0.0160001s, cumul
    0.0300001s
00:00:01.060 Discrete post done after 0.0180001s, cumul 0.167001s
00:00:01.060
Waiting list candidate found.
...
State was added to the waiting list.
...
```

Listing 5.2: *SpaceEx Console Output*

The parser can detect several different results for the computation. The lines containing `Iteration` refer to the start of the analysis of a new state, i.e. the execution of one iteration. Firstly, a time elapse is computed for the location `cycle1_1` as specified in `time elapse in loc(cycleSequence)==cycle1_1`. Once this step is finished, the analysis starts by determining the discrete steps, which the system can take. The intersection of the current state set, i.e. the succesors of the time elapse, and the transition guards are checked. If there is at least one interval in these states, which intersects with the guards, the target

state of the computation can be reached and is added as a possible candidate (`Waiting list candidate found`). The transition which is checked can be found in the line starting with `discrete post`. The information in this line contain the source and target locations and the transition label.

After all possible discrete steps have been checked, the candidate states are either added to the waiting list, i.e. the list of states which has to be examined for new reachable states, or are discarded. A state is added if it has not yet been visited, i.e. the candidate state is transferred to the waiting list (`State was added to the waiting list`). Otherwise, the state is discarded (`State was not added to the waiting list`). During the special case, where the newly reach state is a superset of a state, i.e. the state includes another state, in the waiting list, the new state is merged with waiting list state (`State was merged with another state in the waiting list`). If a state in the waiting list is a superset of the new state the states are merged analogously. The default setting is that the merging of states is enabled, but it can be disabled if necessary.

Analyzing the output file, we can obtain a reachability tree, which does not contain redundant states. The outputs referring to states being added to the waiting list can be used to determine, which states are reached. Thus, a node is added to the tree, if it has been added to the waiting list. We now consider the simplified console output excerpt presented in Listing 5.3, which only shows the time elapses, discrete steps and which states have been added to the candidate and waiting list.

```
time elapse in loc(cycleSequence)==init
discrete post loc(cycleSequence)==init to loc(cycleSequence)==c1_1
Waiting list candidate found.
discrete post loc(cycleSequence)==init to loc(cycleSequence)==c1_2
State was added to the waiting list.

time elapse in loc(cycleSequence)==c1_1
discrete post loc(cycleSequence)==init to loc(cycleSequence)==c2_1
Waiting list candidate found.
discrete post loc(cycleSequence)==init to loc(cycleSequence)==c2_2
Waiting list candidate found.
State was added to the waiting list.
State was added to the waiting list.

time elapse in loc(cycleSequence)==c2_1
discrete post loc(cycleSequence)==init to loc(cycleSequence)==c3_1
Waiting list candidate found.
discrete post loc(cycleSequence)==init to loc(cycleSequence)==c3_2
Waiting list candidate found.
State was added to the waiting list.
State was added to the waiting list.

time elapse in loc(cycleSequence)==c2_2
discrete post loc(cycleSequence)==init to loc(cycleSequence)==c3_1
```

```
Waiting list candidate found.
discrete post loc(cycleSequence)==init to loc(cycleSequence)==c3_2
State was not added to the waiting list.

time elapse in loc(cycleSequence)==c3_1
discrete post loc(cycleSequence)==init to loc(cycleSequence)==c4_1
Waiting list candidate found.
discrete post loc(cycleSequence)==init to loc(cycleSequence)==c4_2
Waiting list candidate found.
State was added to the waiting list.
State was added to the waiting list.

...
```

Listing 5.3: *SpaceEx Console Output*

The tree we can derive from this output starts in the node *init* performing a time step, where `c1_1`, `c1_2`, ... , `c4_2` correspond to locations $c_{1,1}$, $c_{1,2}$, ... , $c_{4,2}$. Afterwards, all discrete transitions are checked. The transition from *init* to $c_{1,1}$ is added to the candidate list. There is no possible intersection of the state after the time elapse with the guard of the transition to $c_{1,2}$ as it is not added to the candidate list. Afterwards, the state reached by the transition from *init* to $c_{1,1}$ is added to the waiting list. Thus, $c_{1,1}$ is added as a successor node of *init*. In the next step $c_{1,1}$ is removed from the waiting list and is analyzed. After the time elapse is performed, the transitions from $c_{1,1}$ to $c_{2,1}$ and $c_{2,2}$ are both added to candidate list and to the waiting list. Therefore, the node $c_{1,1}$ has the two successor nodes $c_{2,1}$ and $c_{2,2}$.

Similarly, the node $c_{2,1}$ receives the two successor nodes $c_{3,1}$ and $c_{3,2}$ as both are added to the waiting list. The waiting list now consists of $c_{2,2}$,$c_{3,1}$ and $c_{3,2}$. The computation of the successors of $c_{2,2}$ detects only one candidate. However this candidate is discarded as the state the automaton would result in has already been reached. Therefore, the node $c_{2,2}$ has no successors in the tree. The last computation we consider is $c_{3,1}$, where both transitions to $c_{4,1}$ and $c_{4,2}$ can be taken and both resulting states are added to the waiting list. We omit the remaining computations for the states for $c_{3,2}$,$c_{4,1}$ and $c_{4,2}$ in the waiting list and assume no new states are found. The resulting reachability tree without the valuations is illustrated in Figure 5.2.

As we are trying to detect whether the locations associated with the final PLC cycle are reachable, we consider the different location paths represented in the tree. This tree contains the path $init \rightarrow c_{1,1} \rightarrow c_{2,2}$, the path $init \rightarrow c_{1,1} \rightarrow c_{2,1} \rightarrow c_{3,2}$, the path $init \rightarrow c_{1,1} \rightarrow c_{2,1} \rightarrow c_{3,1} \rightarrow c_{4,1}$ and the path $init \rightarrow c_{1,1} \rightarrow c_{2,1} \rightarrow c_{3,1} \rightarrow c_{4,2}$.

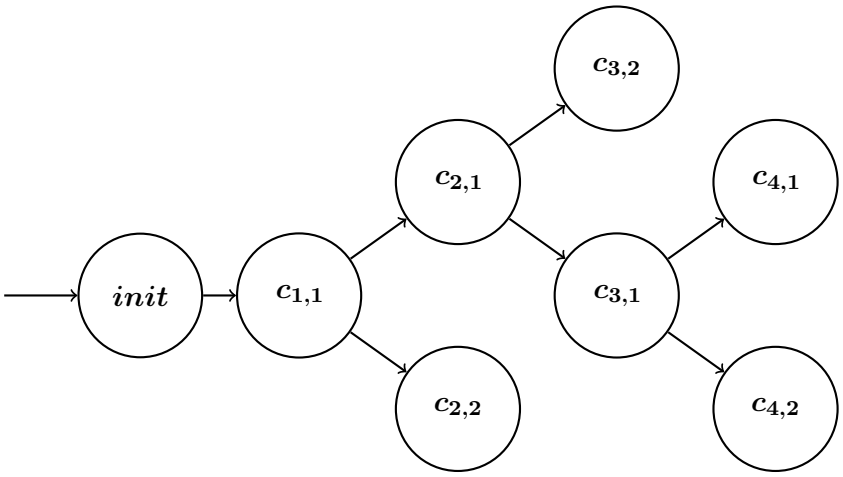

Figure 5.2: *Reachability Tree*

5.2.2 Flow* Reachability Tree

In order to construct a reachability tree from the Flow* output using the current version, forbidden states have to be set for the analysis. Flow* provides a counterexample, which contains a path from an initial state to a forbidden state, if the forbidden state is reached. By defining all states as forbidden sates, the Flow* counterexamples provide all reachable paths. In order to define such forbidden states, they have to intersect with every visited state. This is simply achieved by defining the forbidden states as the set of states where the time t is $t \geq 0$. Each of these paths represents a sequence of states that ends in a forbidden state. As all reached states are forbidden, there are also a multiple prefixes of a path included as for a path with length n all prefix paths with length 1 to $n-1$ are also part of the output. An exemplary counterexample output file excerpt containing only the path information is shown in Listing 5.4.

```
...
computation path
{
singleInitLoc;
}
...
computation path
{
singleInitLoc ( 7 , [0.0000000000e+00 , 6.1035156251e-06]) -> cycle1_1;
}
...
computation path
{
singleInitLoc ( 7 , [0.0000000000e+00 , 6.1035156251e-06]) -> cycle1_1 ( 11 ,
    [8.9999389648e-01 , 1.1000061036e+00]) -> cycle2_1;
}
...
computation path
{
```

```
singleInitLoc ( 7 , [0.0000000000e+00 , 6.1035156251e-06]) -> cycle1_1 ( 11 ,
    [8.9999389648e-01 , 1.1000061036e+00]) -> cycle2_2;
}
...
computation path
{
singleInitLoc ( 7 , [0.0000000000e+00 , 6.1035156251e-06]) -> cycle1_1 ( 11 ,
    [8.9999389648e-01 , 1.1000061036e+00]) -> cycle2_1 ( 37 , [6.9999389648e
    -01 , 1.3000061036e+00]) -> cycle3_1;
}
...
computation path
{
singleInitLoc ( 7 , [0.0000000000e+00 , 6.1035156251e-06]) -> cycle1_1 ( 11 ,
    [8.9999389648e-01 , 1.1000061036e+00]) -> cycle2_2 ( 37 , [6.9999389648e
    -01 , 1.3000061036e+00]) -> cycle3_2;
}
...
computation path
{
singleInitLoc ( 7 , [0.0000000000e+00 , 6.1035156251e-06]) -> cycle1_1 ( 11 ,
    [8.9999389648e-01 , 1.1000061036e+00]) -> cycle2_1 ( 37 , [6.9999389648e
    -01 , 1.3000061036e+00]) -> cycle3_1 ( 59 , [4.9999389648e-01 ,
    9.0001220704e-01]) -> cycle4_1;
}
...
```

Listing 5.4: *Flow* Counterexample Output*

A `computation path` describes a reachable path, where $l_1(n, [a, b]) -> l_2$ describes the jump numbered n from source location l_1 to target location l_2 while the execution time is included in $[a, b]$. In the following we only consider the reachable locations of each path. The counterexample file includes more information about the variable values in each location, which is not required for our transformation. We use the computation paths to construct a reachability tree. We receive a single tree for the given counterexample file, as all paths start in the `singleInitLoc`. In the following we refer to `singleInitLoc` as *init* and to `cycle1_1`, ... , `cycle4_1` as $c_{1,1}$, ... , $c_{4,1}$. The resulting location tree for the example in Figure 5.3.

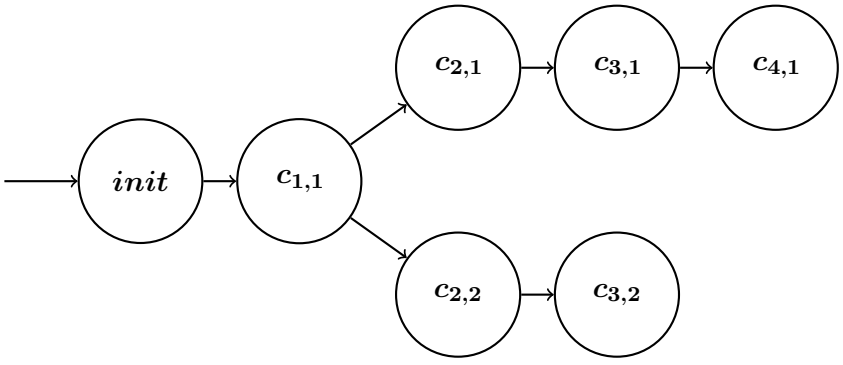

Figure 5.3: *Reachability Tree*

The resulting reachability tree in Figure 5.3 contains two maximum location paths, i.e. paths that start in the root node and end in a leaf node. There is a path $init \rightarrow c_{1,1} \rightarrow c_{2,2} \rightarrow c_{3,2}$ of length four and a path $init \rightarrow c_{1,1} \rightarrow c_{2,1} \rightarrow c_{3,1} \rightarrow c_{4,1}$ of length five. We do not need to consider the shorter paths, as they are prefixes of the two maximum paths, thus they provide only redundant information. The constructed tree can be used for further analysis.

5.3 Explanation Generation

An explanation for a counterexample can be derived by analyzing the reachable paths of the counterexample automaton, which have been computed in the hybrid reachability analysis. In order to generate explanations, we consider all paths starting in a root node and ending in a leaf node. For each of these paths, we must determine the number of PLC cycles that have been visited. If the last cycle is reachable, the counterexample is replicable. Otherwise, by adding the successor of the last reachable cycle which has been reached, we can use this information to construct a prefix sequence of the counterexample, which is unreachable in the hybrid automaton, and generate an explanation. As there are multiple locations that correspond to a single PLC cycle, we have to consider the groupings for these locations as constructed in Section 3.7.1. The automaton may stay in the same PLC but in different locations modeling different conditional ODE systems. This is possible due to the construction of *copy transitions* as presented in Section 3.7.2. These groups can be defined as sets of locations. We consider the reachability tree given in Figure 5.4 for an exemplary path analysis.

The illustrated reachability tree provides six different paths we must consider for the explanation generation. The paths are derived by extracting all paths starting in the root node(s) and ending in a leaf. In this case, there is only a single root node *init*, thus all paths start with *init*. Furthermore, there are six different leaf nodes, namely $c_{3,3}$, $c_{3,1}$, $c_{3,2}$, $c_{3,1}$, $c_{4,1}$ and $c_{4,2}$. The paths resulting from the given reachability tree as explained in Section 2.5.3 for the explanation generation are depicted in Equations (5.4) to (5.9). For now, we only consider the locations for the explanation generation.

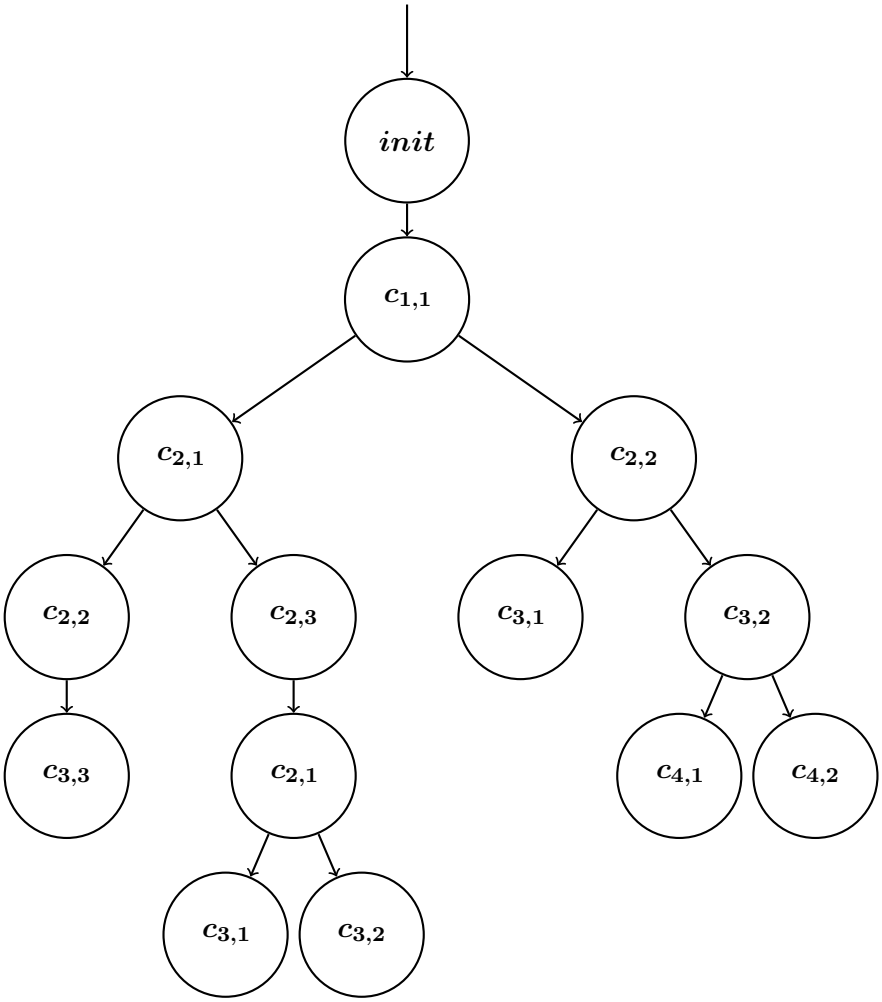

Figure 5.4: *Exemplary Reachability Tree*

$$init \rightarrow c_{1,1} \rightarrow c_{2,1} \rightarrow c_{2,2} \rightarrow c_{3,3} \tag{5.4}$$
$$init \rightarrow c_{1,1} \rightarrow c_{2,1} \rightarrow c_{2,3} \rightarrow c_{2,1} \rightarrow c_{3,1} \tag{5.5}$$
$$init \rightarrow c_{1,1} \rightarrow c_{2,1} \rightarrow c_{2,3} \rightarrow c_{2,1} \rightarrow c_{3,2} \tag{5.6}$$
$$init \rightarrow c_{1,1} \rightarrow c_{2,2} \rightarrow c_{3,1} \tag{5.7}$$
$$init \rightarrow c_{1,1} \rightarrow c_{2,2} \rightarrow c_{3,2} \rightarrow c_{4,1} \tag{5.8}$$
$$init \rightarrow c_{1,1} \rightarrow c_{2,2} \rightarrow c_{3,2} \rightarrow c_{4,2} \tag{5.9}$$

As mentioned before, we are only interested in the PLC cycles that have been visited. The grouping of locations belonging to the same PLC cycle allows us to compute these cycles. In this example we assume the groups $C_1 := \{c_{1,1}, c_{1,2}, c_{1,3}\}$, $C_2 := \{c_{2,1}, c_{2,2}, c_{2,3}\}$, $C_3 := \{c_{3,1}, c_{3,2}, c_{3,3}\}$ and $C_4 := \{c_{4,1}, c_{4,2}, c_{4,3}\}$ as represen-

tatives for the PLC cycles 1, 2, 3, and 4. Thus, three different locations for each PLC cycle were created by adding the conditional ODEs as shown in Section 3.7.1 as each conditional ODE system and their combined negation result in a new location for each PLC cycle.

Determining which PLC cycles have been reached can be achieved by checking the membership of each location in the path. Once at least one member $c_{i,j}$ for $j \in \{1, 2, 3\}$ of a set C_i has been reached in a path, the PLC cycle C_i has been visited. The visited cycles are shown in Equations (5.10) to (5.15) for each path in Equations (5.4) to (5.9).

$$C_1 \to C_2 \to C_3 \tag{5.10}$$
$$C_1 \to C_2 \to C_3 \tag{5.11}$$
$$C_1 \to C_2 \to C_3 \tag{5.12}$$
$$C_1 \to C_2 \to C_3 \tag{5.13}$$
$$C_1 \to C_2 \to C_3 \to C_4 \tag{5.14}$$
$$C_1 \to C_2 \to C_3 \to C_4 \tag{5.15}$$

The longest path in the reachability tree does not have to correspond to the longest PLC cycle sequence as there might be several locations corresponding to the same PLC cycle, which are reachable on the same path. In this example the paths Equations (5.5) and (5.6) are the longest in the original reachability tree, but only visit the PLC cycles C_1, C_2 and C_3, while paths Equations (5.8) and (5.9) are shorter in the original tree, but provide the longest cycle sequence of C_1, C_2, C_3 and C_4.

The explanation generation only considers a longest cycle sequence as it represents the reachable cycles in the hybrid automaton. Due to the sequential construction of this automaton, all shorter cycle sequences are prefixes of the longest sequence. The longest cycle sequence $C_1 \to C_2 \to C_3 \to C_4$, still represents a sequence which is replicable in the counterexample automaton. We add the successor cycle C_5 of the final cycle C_4 in the longest sequence, if the cycle corresponding to the last PLC cycle is not reached, thus we construct a sequence $C_1 \to C_2 \to C_3 \to C_4 \to C_5$ which is no longer replicable. C_5 is not reachable during the hybrid reachability analysis, thus adding the cycle C_5 creates an unreachable cycle sequence. Furthermore, this new sequence is the shortest cycle sequence which can be used to construct an unreachable prefix sequence, since all shorter sequences are still feasible as seen in Equations (5.10) to (5.15). Since all shorter cycle sequences can be omitted, there are two different cases for the longest cycle sequence, which must be handled by the explanation generation.

Considering a counterexample $(\mathit{Vars}, \mathit{Seqs})$ with sequence length n and the longest reachable cycle sequence $C_1 \to C_2 \to \ldots \to C_k$:

1. At least one location corresponding to the last PLC cycle n is reachable in the counterexample automaton, i.e. $k = n$

2. **No** location corresponding to the last PLC cycle n is reachable in the counterexample automaton, i.e. $k < n$

If the last cycle is reached in the hybrid automaton, the counterexample has been validated in the hybrid model. Thus, the counterexample provided by the bounded model checking is still a counterexample given the plant dynamics. The BMC is informed about the result by sending it an empty explanation $(\varnothing, \varnothing)$, i.e., an explanation with empty sequences.

The second possible result is that the last cycle is not reached in any of the paths, which are parsed from the reachability tree. As explained before, we only need to consider the longest cycle sequences resulting from these paths. The sequence describes, which cycles are reachable in the hybrid automaton and as the last cycle has not been reached, the BMC counterexample can not be fully traversed in the hybrid analysis. Assuming cycle k with $k < n$ is the last reachable PLC cycle, thus all incoming transitions of cycle $k+1$ are never taken during the hybrid analysis. Adding the unreachable PLC cycle $k+1$ allows us to construct prefix of the counterexample of length $k+1$. This prefix is an explanation, which can be used to provide a prefix sequence to the BMC, which is unreachable in the hybrid model and can be excluded in further analysis.

In the example with Figure 5.4, the BMC counterexample would be validated, if the counterexample only provided assignments for four cycles and thus paths via the locations that correspond to cycle 4 cannot be prolonged further. The reachability tree analysis then provides us two paths (meaning least one path), which reach the last cycle C_4. Given the BMC counterexample shown in Equations (5.16) to (5.20) and the reachability tree in Figure 5.4, the counterexample is validated, as the last cycle C_4 is reachable.

$$in_{\text{full}} : (0,0,0,1) \tag{5.16}$$
$$in_{\max} : (0,0,1,1) \tag{5.17}$$
$$in_{\min} : (1,1,1,1) \tag{5.18}$$
$$in_{\text{nonempty}} : (1,1,1,1) \tag{5.19}$$
$$out_{\text{v}} : (0,1,1,0) \tag{5.20}$$

An explanation is generated for the reachability tree in Figure 5.4 and the exemplary counterexample in Equations (5.21) to (5.25)

$$in_{\text{full}} : (0,0,0,0,0,0,1) \tag{5.21}$$
$$in_{\text{max}} : (0,0,1,1,0,1,1) \tag{5.22}$$
$$in_{\text{min}} : (1,1,1,1,1,1,1) \tag{5.23}$$
$$in_{\text{nonempty}} : (1,1,1,1,1,1,1) \tag{5.24}$$
$$out_{\text{v}} : (0,1,1,0,0,1,1) \tag{5.25}$$

As seen in Equations (5.10) to (5.15), the longest cycle sequence reaches C_4. Since C_4 is not the last cycle in the original counterexample, it could not be validated given the dynamic behaviour. Extending the sequence using the next cylce, i.e., cycle C_5, we can construct the explanation describing a counterexample prefix of length 5, as this is the shortest sequence which is unreachable in the hybrid model, shown in Equations (5.26) to (5.30).

$$in_{\text{full}} : (0,0,0,0,0) \tag{5.26}$$
$$in_{\text{max}} : (0,0,1,1,0) \tag{5.27}$$
$$in_{\text{min}} : (1,1,1,1,1) \tag{5.28}$$
$$in_{\text{nonempty}} : (1,1,1,1,1) \tag{5.29}$$
$$out_{\text{v}} : (0,1,1,0,0) \tag{5.30}$$

The bounded model checker is able to parse the files, evaluate the explanations and uses them to exclude counterexamples with the prefix provided by the explanations.

5.4 Explanation Processing

The bounded model checking uses logic formulas with respect to the given background theory as presented in Section 2.3 to model the control flow and uses SMT solving to determine the possible execution of the system. Due to the control flow being modeled as logic formulas internally, the BMC can remember the current state of the system by storing the corresponding formula. If a counterexample is detected, the BMC stores the last states, i.e. the states before the counterexample was found. By retaining the previous states, the analysis can be extended at the appropriate iteration by excluding counterexamples and be resumed instead of being restarted.

Afterwards, a new counterexample is passed to the hybrid analysis and an automaton is constructed, if a counterexample can be found. A hybrid reachability analysis is performed and an explanation is generated. The explanations, which

are constructed from the hybrid reachability analysis, represent sequences that do not describe a counterexample if the plant dynamics are added to the model. If no new BMC counterexample is found, the model is safe.

Assuming, the BMC receives an explanation $(\mathit{Vars}, \mathit{Seqs})$ of equi-length sequences Seqs with length n with the variables $\mathit{var}_i \in \mathit{Vars}$ with $i \in \{1, \ldots, m\}$ as shown in Equations (5.1) to (5.3). Thus set of sequences defines sequences of assignments for the discrete variables, that can not occur if the dynamic behaviour is considered. These sequences represent prefixes of the sequences, which the BMC can exclude. The stored BMC state is recalled and extended by an exclusion formula to avoid the occurrence of the sequences of the given explanation. The exclusion formula is build to exclude the sequence of variable assignments provided by the explanation. Thereafter, the exclusion formula is added in conjunction with the stored BMC state, thus preventing the BMC analysis from reaching the discrete sequence provided by the explanation.

Considering the tank example presented in Section 2.2, which has been discussed before, we have four input variables and one output variable. Each explanation for the tank example are a sequence of assignments for sensors in_{full}, in_{max}, in_{min} and in_{nonempty} and valve out_v. The expressiveness of an explanation decreases with increasing length. If we compare the two explanations for the tank system presented in Equations (5.31) to (5.35) and Equations (5.36) to (5.40), the different formulas to exclude these sequences also show the different expressiveness of the explanations.

$$in_{\text{full}} : (0, 0, 0) \tag{5.31}$$
$$in_{\text{max}} : (0, 0, 1) \tag{5.32}$$
$$in_{\text{min}} : (1, 1, 1) \tag{5.33}$$
$$in_{\text{nonempty}} : (1, 1, 1) \tag{5.34}$$
$$out_{\text{v}} : (1, 1, 1) \tag{5.35}$$

The first explanation given in Equations (5.31) to (5.35) excludes all sequences, where the formula constructed by the bounded model checker ensures, that in the variable assignments 0, 0, 0 for in_{full} and 0, 0, 1 for in_{max} and 1, 1, 1 for in_{min}, in_{far} and out_{v} can not occur for the cycles 1, 2 and 3, respectively. Thus, this excludes all counterexamples beginning with this sequence.

$$in_{\text{full}} : (0) \tag{5.36}$$
$$in_{\text{max}} : (0) \tag{5.37}$$
$$in_{\text{min}} : (1) \tag{5.38}$$
$$in_{\text{nonempty}} : (1) \tag{5.39}$$
$$out_{\text{v}} : (1) \tag{5.40}$$

The explanation shown in Equations (5.36) to (5.40) excludes even more sequences. All sequences where 0 is assigned to in_{full} and in_{max} and 1 is assigned to in_{min}, in_{nonempty} and out_{v} in the first cycle are no longer viable for the BMC after the appropriate formula has been added.

Therefore, after each explanation processed by the BMC, the control flow is extended by an exclusion formula. The BMC analysis is then resumed with the new control flow model.

In the second case, when the BMC does not receive an explanation, the BMC can stop its analysis, as the hybrid analysis has confirmed the counterexample. Due to the undecidability of the hybrid reachability, we cannot consider the model *unsafe*, as the analysis might have over-approximated the reachable states. The result for the analysis is *unknown* as either an actual hybrid counterexample has been detected or the hybrid analysis is too inaccurate.

5.5 Explanation Wildcards

Wildcards as introduced in Section 3.8 can also be used to improve the expressiveness of explanations. As shown in Section 5.3, the explanations we derive from the hybrid analysis are prefixes of the given counterexamples. These prefixes allow us to exclude counterexamples with the same initial sequences as explained in Section 5.4.

The addition of wildcards to an explanation allows us to tell the BMC to exclude specific variable assignments at specific cycles. Each wildcard should be considered by using all possible assignments for the variable in the construction of an exclusion formula as described in Section 5.4, as they can take any values for the given data type. The explanations still retain their prefix property when wildcards are used. The expressiveness of an explanation is increased with an increasing amount of wildcards, as more counterexamples will be excluded.

As we only consider boolean wildcards, the expressiveness change of an explanation for each occurrence of a wildcard assignment is that the set of counterexample sequences which can be excluded is doubled. Assuming we have an explanation for a system with a single variable in_{var} with a single boolean wildcard as shown in Equation (5.41), we exclude 2 prefixes instead of just 1.

$$in_{\text{var}} : (1, *, 0, 1) \tag{5.41}$$

The given explanation excludes the prefixes $(1, 0, 0, 1)$ and $(1, 1, 0, 1)$ for in_{var}. Each additional wildcard would double the amount of excludable prefixes, as the prefixes with all possible assignments $\{0, 1\}$ for each wildcard can be excluded.

An exemplary explanation with wildcards for the tank example, which has been presented in Section 2.2 is given in Equations (5.42) to (5.46).

$$in_{\text{full}} : (*, 0, *, 1, *) \tag{5.42}$$
$$in_{\text{max}} : (*, *, *, *, *) \tag{5.43}$$
$$in_{\text{min}} : (*, *, *, *, *) \tag{5.44}$$
$$in_{\text{nonempty}} : (*, *, *, *, *) \tag{5.45}$$
$$out_{\text{v}} : (*, *, *, *, *) \tag{5.46}$$

The explanation results in the exclusion of all sequences, where in_{full} is assigned 0 in the second cycle and 1 in the fourth cycle. Unfortunately, the wildcards still do not allow us to exclude the occurrences of specific subsequences in the discrete counterexamples, as the we can only define a wildcard for a specific cycle. Considering the single variable system with variable in_{var}, we can exclude an exemplary subsequence $(0, 1, 0)$ at the beginning by constructing the explanation given in Equation (5.47).

$$in_{\text{var}} : (0, 1, 0) \tag{5.47}$$

If we want to exclude the subsequence starting at any other position than the first cycle, we have to construct new explanations as shown in Equations (5.48) to (5.50) which represent the exclusion of the subsequence starting in cycle 2, 3 and 4.

$$in_{\text{var}} : (*, 0, 1, 0) \tag{5.48}$$
$$in_{\text{var}} : (*, *, 0, 1, 0) \tag{5.49}$$
$$in_{\text{var}} : (*, *, *, 0, 1, 0) \tag{5.50}$$

Currently the BMC is able to process explanation wildcards. However we still require heuristics to determine when a wildcard can be added in an explanation.

5.6 Summary

In this chapter we have presented how the results from a hybrid reachability analysis can be used to construct explanations, which are used to exclude discrete counterexamples that are not counterexamples if the dynamic behaviour is considered. Starting out with the analysis of the verification console output of SpaceEx, we show how to construct a reachability tree. The console output is used to determine, which states are added to the tree and which are omitted. Additionally, the Flow* counterexample output is used to construct a reachability

tree. Each counterexample path of the Flow* counterexamples and its prefixes represent a path from the root to a leaf in the tree. Examples are provided for both verification tools. A method is introduced to generate an explanation for an exemplary reachability tree. We discussed the expressiveness of these explanations on the basis of exemplary explanations. The functionality of the BMC is presented in respect to the processing of explanations. The BMC remembers the previous states in order resume its analysis after an explanation has been used to exclude specific subsequences from further analysis. Moreover, we also provide a possibility to improve the expressiveness of explanations by allowing wildcards. However, these wildcards still have some restrictions.

Chapter 6

Experimental Results

In this chapter, the presented verification approach is applied to two different systems. The evaluations of the verifications are used to determine weaknesses in the current algorithms as well as to show how the hybrid reachability affects the results of the analysis. Moreover, special cases which might occur during the analysis are discussed. A detailed analysis of the runtime of each system is provided as well.

We present the models for the tank system and a new system describing a train crossing in Section 6.1. In addition to the tank system as defined for our new approach, we also provide a model for the entire system in order to compare the analyses. The verification process of the different systems and problems which arose during this process are discussed in Section 6.2. Finally, a runtime analysis for the verification is performed in Section 6.3. We consider the two systems as well as manually generated counterexample sequences of different lengths for both systems in order to analyze the effects of increasing counterexample lengths on the hybrid analysis. Furthermore, we provide some improvements for the hybrid analysis.

6.1 Exemplary Systems

In this section we present two different systems which are controlled by PLCs. The first system in Section 6.1.1 is the tank system which has been introduced in Section 2.2. Furthermore, we introduce a new system in Section 6.1.2, which models a train crossing. Gates have to be lowered if the train passes the crossing and are raised if the train has left the crossing. We define these systems in the following and show how to model their plant dynamics.

6.1.1 Tank System

The tank system has already been described in detail in the previous chapters Chapter 2 and Chapter 4. We assume a tank with four water height sensors and a valve, which controls the water level change in the tank. Furthermore, there is a hole at the bottom of the tank, which allows water to leak out of the tank. An illustration of this tank system is shown in Figure 6.1.

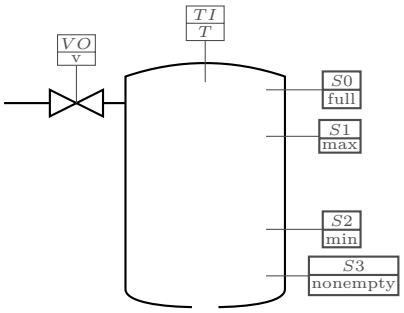

Figure 6.1: *Tank System*

The input parameters for the system consist of replacement rules for the discrete variables, conditional ODEs and initial values. Furthermore, we have to define a cycle time and configure the SpaceEx parameters appropriately. The PLC program, which is analyzed by the bounded model checker is the same as in Listing 2.1. The intermediate verification language models the discrete behaviour of the tank system.

In the tank example, the replacement rules define the water levels at which each sensor is activated. The dynamic links for the discrete boolean variables are defined in Table 6.1.

Variable	*true*	*false*
in_{full}	$h \geq 20$	$h < 20$
in_{max}	$h \geq 15$	$h < 15$
in_{min}	$h \geq 5$	$h < 5$
in_{nonempty}	$h \geq 0$	$h < 0$

Table 6.1: *Variable Links*

The conditional ODEs define the water flow according the the output variable out_v. For this analysis we use the conditional ODEs in Equations (6.1) and (6.2).

$$out_v == 1 : \dot{h}_1 = 2, \tag{6.1}$$

$$out_v == 0 : \dot{h}_1 = -2 \tag{6.2}$$

The initial values of h depend on the initial sensor assignments. There are five different areas in the tank, which are described by the sensors. The conditional initial values are depicted in Equations (6.3) to (6.7).

$$in_{\text{full}} : h := 25 \tag{6.3}$$
$$\neg in_{\text{full}} \wedge in_{\text{max}} : h := 17 \tag{6.4}$$
$$\neg in_{\text{max}} \wedge in_{\text{min}} : h := 10 \tag{6.5}$$
$$\neg in_{\text{min}} \wedge in_{\text{nonempty}} : h := 4 \tag{6.6}$$
$$\neg in_{\text{nonempty}} : h := -1 \tag{6.7}$$

The input files for the given input parameters are described in Chapter 4. We assume a cycle time of exactly 1 and configure SpaceEx accordingly. The configurations in Listing 6.1 show the SpaceEx parameters, which are used for the hybrid reachability analysis.

```
system = system
forbidden = global_time < 0
output-variables = global_time,h

scenario = supp
directions = box
sampling-time = 0.01
time-horizon = 1
iter-max = -1
output-format = GEN
verbosity = m
rel-err = 1.0e-12
abs-err = 1.0e-13
```

Listing 6.1: *SpaceEx Parameters*

The main component of the model is called `system`. The forbidden states have to be assigned, but are simply deactivated by making them unreachable. In this case we have a time variable `global_time`, which is always positive, thus setting the forbidden states as all states where `global_time` is negative, defines states which are never reached. The forbidden states are deactivated as SpaceEx only returns state information restricted to the forbidden states if forbidden states are reached. We use the LGG support function scenario with a `sampling-time` of 0.01, i.e. a fixed step size of 0.01. The time-horizon of 1 is sufficient as we have a cycle time of 1. Furthermore, the maximum iterations that are performed by SpaceEx can be set as infinite due to the sequential structure, the conditional ODE and cycle time restrictions of the automaton. The analysis will always stop after a reasonable amount of time either to the time restrictions or the transition guards.

Tank System Full Automaton

We also consider a hybrid automaton modeling the tank system as a combination of the entire discrete behaviour and all plant dynamics. We omit the assignments for the discrete variables, as they can be modeled by the location and dynamic guards. The hybrid automaton is illustrated in Figure 6.2.

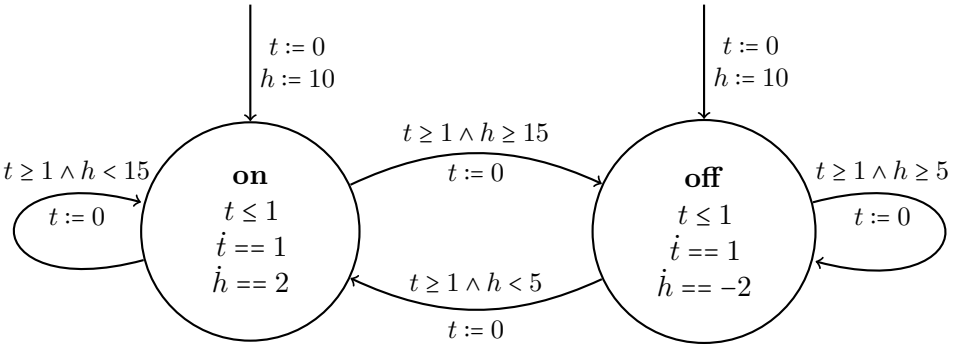

Figure 6.2: *Tank System Hybrid Automaton*

The automaton contains two locations describing the state of the valve. In **on** the valve is open and the water is rising $\dot{h} == 2$ and in **off** the valve is closed and the water level is falling $\dot{h} == -2$. Furthermore, both locations have a cycle timer t, which causes the system to stay in each location for the constant cycle time of 1. This is accomplished by adding the invariant $t \leq 1$ to all locations and the guard $t \geq 1$ to all transition guards, while time passes at a constant value of $\dot{t} == 1$. As the system may only stay in the same location, considering conditions over the water level h, after a PLC cycle has been executed, all transition guards are extended accordingly.

The transition from **on** to **off** can only be taken if the $in_{\max}$ is active. As defined in the replacement rules, this translates to the condition $h \geq 15$. Similarly, the guard of transition **off** to **on** is extended by $h < 5$ as it should only be taken, if $in_{\min}$ is deactivated. The self loops of the locations are extended with the guards $h \geq 5$ and $h < 15$ respectively, thus not allowing the system to stay in **on** if the $in_{\max}$ is active and forcing the system to leave **off** if the $in_{\min}$ sensor is deactivated. We assume, that the system starts between $in_{\min}$ and $in_{\max}$ and initialize h with an exemplary value of 10 or with the entire area $[5, 15)$.

The unsafe states for the system are defined as $h < 0 \vee h \geq 20$ as these are the states, where in_{full} is active or in_{nonempty} is not active. The unsafe states are also derived from the replacement rules.

6.1.2 Train Crossing

In the second example we apply our PLC program verification approach to a model of a train crossing. The train tracks cross a street with a gate on each side of the street to prevent cars from crossing. The example only considers the behaviour of the train and the gates. The PLC controls the gates of the train crossing by raising and lowering them according to the train position. When the train is near the crossing, the gates start lowering. After the train has passed the crossing, the gates open again. The schematic shown in Figure 6.3 illustrates the layout of the sensors and gates used by the controller.

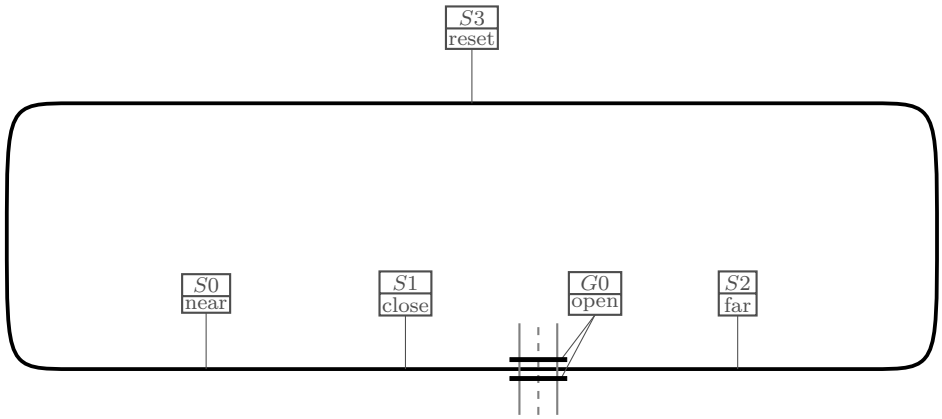

Figure 6.3: *Train Crossing*

In the example, the sensors in_{near}, in_{close}, in_{far} and in_{reset} are activated, if the train passes over them, and are deactivated once the succeeding sensor on the track is passed. The output variables out_{posmsb} and out_{poslsb} define different areas of the track. Thus, the assignment of $out_{\text{posmsb}} := 0$ (most significant bit) and $out_{\text{poslsb}} := 0$ (least significant bit), i.e., 00, defines the track between the sensors in_{near} and in_{close}. Furthermore, the assignments 01, 10 and 11 define the track areas between in_{close} and in_{far}, in_{far} and in_{reset} and in_{reset} and in_{near}, respectively. The sensor in_{open} defines the state of the gates on both sides of the crossing. Therefore, the condition in_{open} describes opened gates, while $\neg in_{\text{open}}$ describes closed gates.

The PLC program is given as an instruction list for the bounded model checker. Assumptions in the program restrict the system in such a manner, that exactly one of the position sensors (in_{near}, in_{close}, in_{far} and in_{reset}) is active at any time. The unsafe states for the system are where the sensor in_{open} is *true*, while in_{close} is *true*. Furthermore, if in_{reset} is active, the gates have to be closed ($\neg in_{\text{open}}$) for the system to be in a safe state.

In addition to the discrete behaviour, we define two new variables p and g. The continuous variable p describes the current position of the train, while g models the angle of the train crossing gates. Thus, $g = 0$ defines fully closed gates and $g = 90$ opened gates. There are two groups of dynamic links. The first group connects the position sensors with the position p of the train, while the second group describes the state of the gates and the corresponding dynamic conditions. It is sufficient to model only the links for each activated position sensor as the positon conditions for p do not intersect. The links for the position sensors are shown in Figure 6.4.

Variable	*true*
in_{near}	$p \geq 1 \wedge p < 4$
in_{close}	$p \geq 4 \wedge p < 6$
in_{far}	$p \geq 6 \wedge p < 10$
in_{reset}	$p \geq 10,\ p := 0$

Figure 6.4: *Position Sensor Links*

Two neighbouring sensors describe an area for the train position p. Thus, each sensor stays active while the train is inside a specific area. If the train passed sensor in_{near}, its position should be $p \in [1, 4)$. The sensors in_{close} and in_{far} describe the train positions $p \in [4, 6)$ and $p \in [6, 10)$, respectively. All other possible train positions $p \geq 10$ are described by in_{reset} as the sensor activation resets the actual train position p to 0, thus allowing the train to travel back to the first position sensor.

The train crossing gates have two states. They are either closed if the angle g is $g \leq 10$. All other values of $g > 10$ correspond to open gates. The replacement rules for the in_{open} are defined in Figure 6.5.

Variable	*true*	*false*
in_{open}	$g > 10$	$g \leq 10$

Figure 6.5: *Position Sensor Links*

By combining the replacement rules of the gate sensor with the rules of the position sensors, we have modeled all sensors of the train crossing system. All the dynamic links are defined as described in Section 4.1.2. The link file containing all replacement rules is defined in Listing 6.2.

```
epsilon = 0.00001
in_reset == 1 <=> p >= 10 <=> p:=0
in_near == 1 <=> p >= 1 AND p < 4
in_close == 1 <=> p >= 4 AND p < 6
in_far == 1 <=> p >= 6 AND p < 10
in_open == 1 <=> g > 10
in_open == 0 <=> g <= 10
```

Listing 6.2: *Train Crossing Link File*

The different continuous behaviours of p and g are given by conditional ODEs. The conditions of these ODEs depend on the output variables out_{posmsb} and out_{poslsb} and the angle of the gates g. In order to model the dynamic behaviour, we define six conditional ODEs as depicted in Equations (6.8) to (6.13).

$$out_{\text{posmsb}} \wedge out_{\text{poslsb}} : \dot{p} == 1, \dot{g} == 0 \tag{6.8}$$

$$out_{\text{posmsb}} \wedge \neg out_{\text{poslsb}} \wedge g \leq 90 : \dot{p} == 1, \dot{g} == 45 \tag{6.9}$$

$$out_{\text{posmsb}} \wedge \neg out_{\text{poslsb}} \wedge g \geq 90 : \dot{p} == 1, \dot{g} == 0 \tag{6.10}$$

$$\neg out_{\text{posmsb}} \wedge out_{\text{poslsb}} : \dot{p} == 1, \dot{g} == 0 \tag{6.11}$$

$$\neg out_{\text{posmsb}} \wedge \neg out_{\text{poslsb}} \wedge g \geq 0 : \dot{p} == 1, \dot{g} == -45 \tag{6.12}$$

$$\neg out_{\text{posmsb}} \wedge \neg out_{\text{poslsb}} \wedge g \leq 0 : \dot{p} == 1, \dot{g} == 0 \tag{6.13}$$

As described before, the combination of the variable assignments of out_{posmsb} and out_{poslsb} define the different areas of the tracks. The train always moves with a constant speed of $\dot{p} == 1$ for each time unit. Thus, the train position change is independent of the current variable assignments. The only changes of g occur, if either the in_{near} or the in_{far} sensor has been passed. This results in either in the assignment $out_{\text{posmsb}} := 0$ and $out_{\text{poslsb}} := 0$ or $out_{\text{posmsb}} := 1$ and $out_{\text{poslsb}} := 0$, which model the opening and closing phases of the gates. Furthermore, the changes of g are restricted by g itself. The angle described by g may never be bigger than 90 or smaller than 0. These conditions result in the two conditional ODEs for each of the output variable assignments. As the conditions have to overlap the values 0 and 90 of g, we define the additional conditions $g \geq 90$, $g \leq 90$, $g \geq 0$ and $g \leq 0$ in the conditional ODEs. The conditions have to overlap for the hybrid reachability to be able to switch between the locations.

The initial values for p and g will be defined according to the position sensors and the gate sensors, respectively. For p we can assume, that at least one position sensor is active, as otherwise the train is not at any specified position. We define the following conditional initial values as shown in Equations (6.14) to (6.17) for p.

$$in_{\text{near}} : g := 0 \tag{6.14}$$
$$in_{\text{close}} : g := 2 \tag{6.15}$$
$$in_{\text{far}} : g := 5 \tag{6.16}$$
$$in_{\text{reset}} : g := 7 \tag{6.17}$$

The gate angle g is initialized according to the in_{open} sensor. The assignments are shown in Equations (6.18) and (6.19).

$$in_{\text{open}} : g := 90 \tag{6.18}$$
$$\neg in_{\text{open}} : g := 0 \tag{6.19}$$

The conditionial ODEs and conditional initial values are stored in an XML file (Listing 6.3) as described in Section 4.1.1.

```
<?xml version="1.0" encoding="UTF-8"?>
<condODEsys>
  <addNegatedTerms>false</addNegatedTerms>
    <condODE>
      <cond><![CDATA[out_posmsb AND out_poslsb]]></cond>
      <equation>p' == 1</equation>
      <equation>g' == 0</equation>
    </condODE>
    <condODE>
      <cond><![CDATA[out_posmsb AND !out_poslsb AND g <= 90]]></cond>
      <equation>p' == 1</equation>
      <equation>g' == 45</equation>
    </condODE>
    <condODE>
      <cond><![CDATA[out_posmsb AND !out_poslsb AND g >= 90]]></cond>
      <equation>p' == 1</equation>
      <equation>g' == 0</equation>
    </condODE>
    <condODE>
      <cond><![CDATA[!out_posmsb AND out_poslsb]]></cond>
      <equation>p' == 1</equation>
      <equation>g' == 0</equation>
    </condODE>
    <condODE>
      <cond><![CDATA[!out_posmsb AND !out_poslsb AND g >= 0]]></cond>
      <equation>p' == 1</equation>
      <equation>g' == -45</equation>
    </condODE>
    <condODE>
      <cond><![CDATA[!out_posmsb AND !out_poslsb AND g <= 0]]></cond>
      <equation>p' == 1</equation>
      <equation>g' == 0</equation>
```

```
        </condODE>
        <init>
            <variable var = "p">
                <condInit>
                    <cond><![CDATA[in_reset]]></cond>
                    <value>0</value>
                </condInit>
                <condInit>
                    <cond><![CDATA[in_near]]></cond>
                  <value>2</value>
              </condInit>
                <condInit>
                    <cond><![CDATA[in_close]]></cond>
                    <value>5</value>
                </condInit>
                <condInit>
                    <cond><![CDATA[in_far]]></cond>
                    <value>7</value>
                </condInit>
            </variable>
            <variable var = "g">
                <condInit>
                    <cond><![CDATA[in_open]]></cond>
                    <value>90</value>
                </condInit>
                <condInit>
                    <cond><![CDATA[NOT in_open]]></cond>
                  <value>0</value>
              </condInit>
            </variable>
        </init>
    </condODEsys>
```

Listing 6.3: *Conditional ODEs and Initial Values*

Further configurations include a constant cycle time of 1. After the given system has been transformed into an hybrid automaton as described in Chapter 3, we use SpaceEx to perform a hybrid reachability analysis.

6.2 Analysis Execution

In this section we present some analyses we have performed on the tank system as presented in Section 6.1.1 and the train crossing as introduced in Section 6.1.2. We discuss the results of the verification of the tank system in Section 6.2.1. Furthermore, we discuss problems which may occur due to over-approximation in the hybrid reachability analysis. In Section 6.2.2, we present the verification of the train crossing example. We show how Zeno behaviour affects the results of the hybrid analysis and how to avoid these problems.

6.2.1 Tank System Analysis

For the tank system presented in Section 6.1.1, we discuss the analysis of 10 PLC cycles in Section 6.2.1 and show how an exemplary counterexample, which is generated during the BMC analysis, is refuted due to the discrete not corresponding the dynamic behaviour. Furthermore, in Section 6.2.1 we show how over-approximation during the hybrid reachability analysis might cause the hybrid analysis to falsely terminate the verification. The last system as we present and analyze in Section 6.2.1 contains an error in the control program, which causes the verification to fail.

Tank System 10 Cycles

Considering the path length 10 , i.e. the number of cycles which are analyzed, during the BMC analysis, the verification of the system returns the result *safe*. As the BMC analysis constructs the counterexamples with increasing length, we know that if a path with length n has been generated as a counterexample, all path with length $k < n$ are safe. We require this restriction as the BMC only allows us to bound the search depth in the CFA and not the actual amount of analyzed PLC cycles.

The entire analysis for a verification of the tank system with at most 10 PLC cycle executions requires 264 iterations. An iteration consists of an execution of the BMC, a counterexample generation, the hybrid analysis of the counterexample and an explanation generation. Furthermore, it includes the exclusion of counterexamples according to the generated explanation during subsequent BMC iterations. After 264 iterations, all discrete counterexamples of length 10 have been refuted due to the hybrid system behaving differently.

We show how the hybrid analysis refutes the counterexamples provided by the BMC, by considering an exemplary counterexample which is generated during the analysis. The counterexample is depicted in Equations (6.20) to (6.24).

$$in_{\text{full}} : (0,0,0,0,0,0,0,0,0,1) \tag{6.20}$$
$$in_{\text{max}} : (0,0,0,0,0,0,0,0,0,1) \tag{6.21}$$
$$in_{\text{min}} : (1,1,1,0,0,0,1,1,1,1) \tag{6.22}$$
$$in_{\text{nonempty}} : (1,1,1,1,1,1,1,1,1,1) \tag{6.23}$$
$$out_{\text{v}} : (0,0,0,0,1,1,1,1,1,1) \tag{6.24}$$

In this counterexample, the discrete analysis has found an unsafe cycle sequence, where the tank is overflowing, i.e., in_{full} is active in a cycle. If we now model

the dynamic behaviour according the the sensor activity and the given plant dynamics, the result shows, that this cycle sequence is not possible. Figure 6.6 illustrates the discrete and dynamic behaviour for the given counterexample.

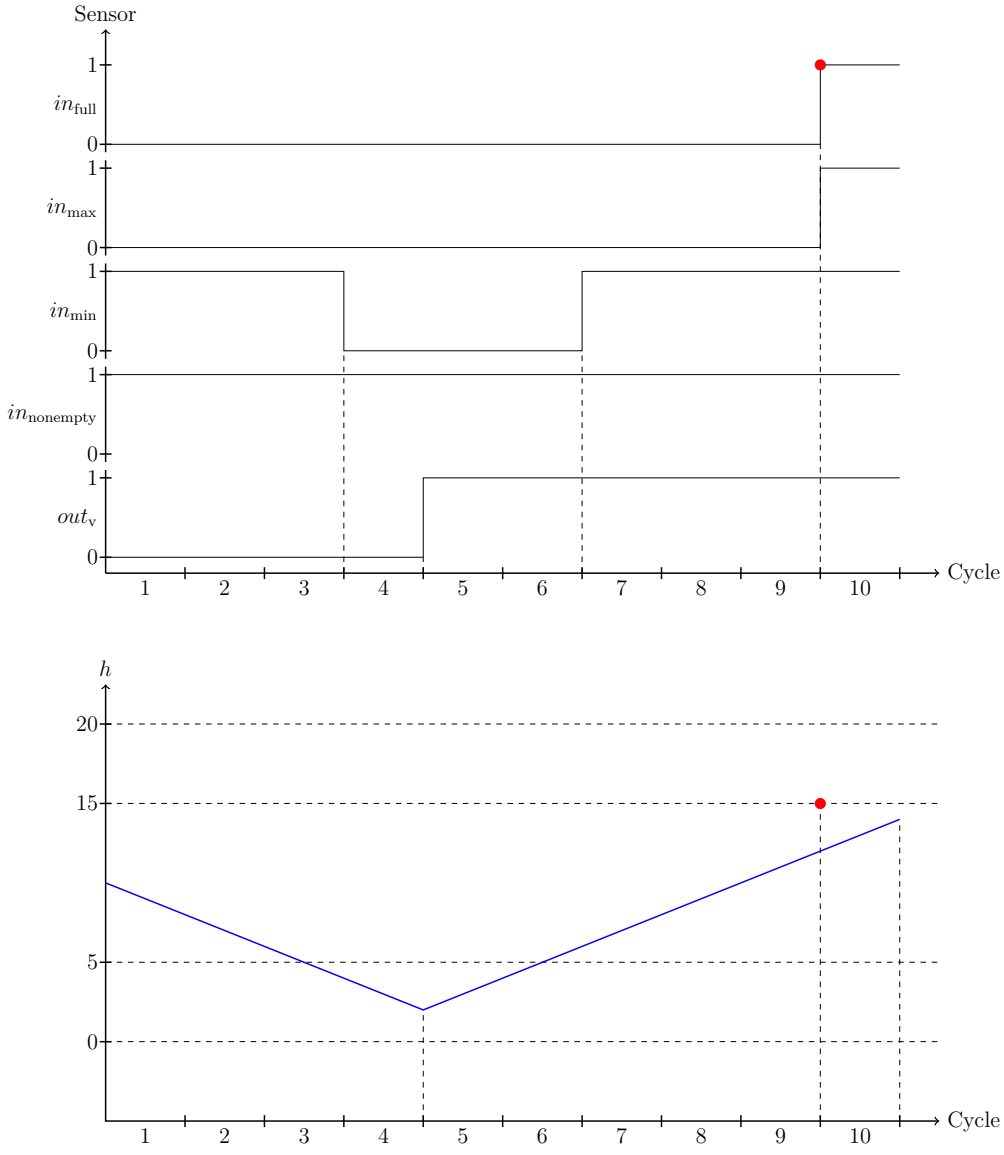

Figure 6.6: *Discrete And Dynamic Behaviour*

The diagrams in Figure 6.6 show, that the dynamic behaviour after cycle 9 differs from the discrete behaviour as marked by the red dots. When entering cycle 10 the sensors in_{max} and in_{full} are defined as *true* by the counterexample, however the value of h is $h < 15$. Thus, when the counterexample automaton tries to enter the locations corresponding to 10th PLC cycle, the analysis stops,

as the replacement rule $in_{\text{max}} \Leftrightarrow h \geq 15$ is applied to all incoming transitions of the cycle 10 locations. Furthermore, $in_{\text{full}} \Leftrightarrow h \geq 20$ are also applied to the same transitions. Either of these two guards stops the hybrid reachability analysis from reaching the last PLC cycle. The explanation, which is generated is shown in Equations (6.25) to (6.29).

$$in_{\text{full}} : (0,0,0,0,0,0,0,0,0) \tag{6.25}$$
$$in_{\text{max}} : (0,0,0,0,0,0,0,0,0) \tag{6.26}$$
$$in_{\text{min}} : (1,1,1,0,0,0,1,1,1) \tag{6.27}$$
$$in_{\text{nonempty}} : (1,1,1,1,1,1,1,1,1) \tag{6.28}$$
$$out_{\text{v}} : (0,0,0,0,1,1,1,1,1) \tag{6.29}$$

Currently, there are still some issues with the BMC analysis, which do not allow us to analyze larger counterexamples. The runtime as well as the memory usage increases drastically during the analysis of longer execution paths. The runtime and memory usage increases with increasing counterexample path length. In Section 6.3.1 we show the increase in runtime and memory usage for longer cycle sequences as measured during the analysis.

Additionally we consider the analysis, where we modify the conditional initial values to represent all possible values for each area of the tank. The analysis is performed for the conditional initial values shown in Equations (6.30) to (6.34).

$$in_{\text{full}} : \; h \coloneqq [20,50] \tag{6.30}$$
$$\neg in_{\text{full}} \wedge in_{\text{max}} : \; h \coloneqq [15,20) \tag{6.31}$$
$$\neg in_{\text{max}} \wedge in_{\text{min}} : \; h \coloneqq [5,15) \tag{6.32}$$
$$\neg in_{\text{min}} \wedge in_{\text{nonempty}} : \; h \coloneqq [0,5) \tag{6.33}$$
$$\neg in_{\text{nonempty}} : \; h \coloneqq [-30,0) \tag{6.34}$$

The new conditional initial values affect the number of iterations required to verify the tank system. The analysis performs 2174 iterations to verify 10 PLC cycles. This increase in iterations occurs since more states are now reachable during the hybrid reachability analysis and thus less counterexamples are excluded during each iteration.

Tank System Approximation Problem

Considering slightly adapted plant dynamics for the tank system, we show a problem which might occur due to over-approximation during the hybrid

reachability analysis. The conditional ODE systems are changed, so that the water drains faster from the tank when the valve is closed. The new conditional ODE systems are shown in Equations (6.35) and (6.36).

$$out_v == 1 : \dot{h}_1 = 2, \tag{6.35}$$

$$out_v == 0 : \dot{h}_1 = -3 \tag{6.36}$$

If the valve out_{v} is closed, the water decreases by 3 for each time unit. We perform an analysis on the system with all other plant dynamics as described in Section 6.1.1. If more than 10 cycles are analyzed, the BMC generates a counterexample which should not be replicable by the hybrid analysis, but the last cycle locations are reachable. This problem occurs due to the over-approximation in the hybrid reachability analysis. The counterexample, where this problem occurs is depicted in Equations (6.37) to (6.41).

$$in_{\mathrm{full}} : (0,0,0,0,0,0,0,0,0,0,0) \tag{6.37}$$

$$in_{\mathrm{max}} : (0,0,0,1,1,1,0,0,0,0,0) \tag{6.38}$$

$$in_{\mathrm{min}} : (1,1,1,1,1,1,1,1,1,0,0) \tag{6.39}$$

$$in_{\mathrm{nonempty}} : (1,1,1,1,1,1,1,1,1,1,0) \tag{6.40}$$

$$out_{\mathrm{v}} : (1,1,1,1,0,0,0,0,0,0,1) \tag{6.41}$$

If we compare the discrete with the dynamic behaviour, we can see that the last cycle is not reachable, as all input sensors are deactivated. Thus, considering the replacement rules in Table 6.1, the applied rules result in the guard $h < 0$ for the incoming transitions of the locations corresponding to the last PLC cycle. Figure 6.7 shows that the discrete behaviour cannot occur as the continuous variables behave differently.

After the execution of the 10th cycle, the value of h is at exactly 0, thus $h < 0$ is not satisfied. However, due to over-approximation, $h < 0$ may be satisfiable during the hybrid reachability analysis. In SpaceEx such problems might occur, if we use the LGG or STC scenario, which over-approximate the reachable values of h as shown in Figure 6.8, which shows the reachable values of h computed by the LGG scenario. As the hybrid automaton produced by the automaton generation is a linear hybrid automaton, we can also apply the PHAVer scenario. The scenario computes exact values for h and as a consequence the system does not reach the last cycle after the reachable values of h have been computed exactly as shown in Figure 6.9.

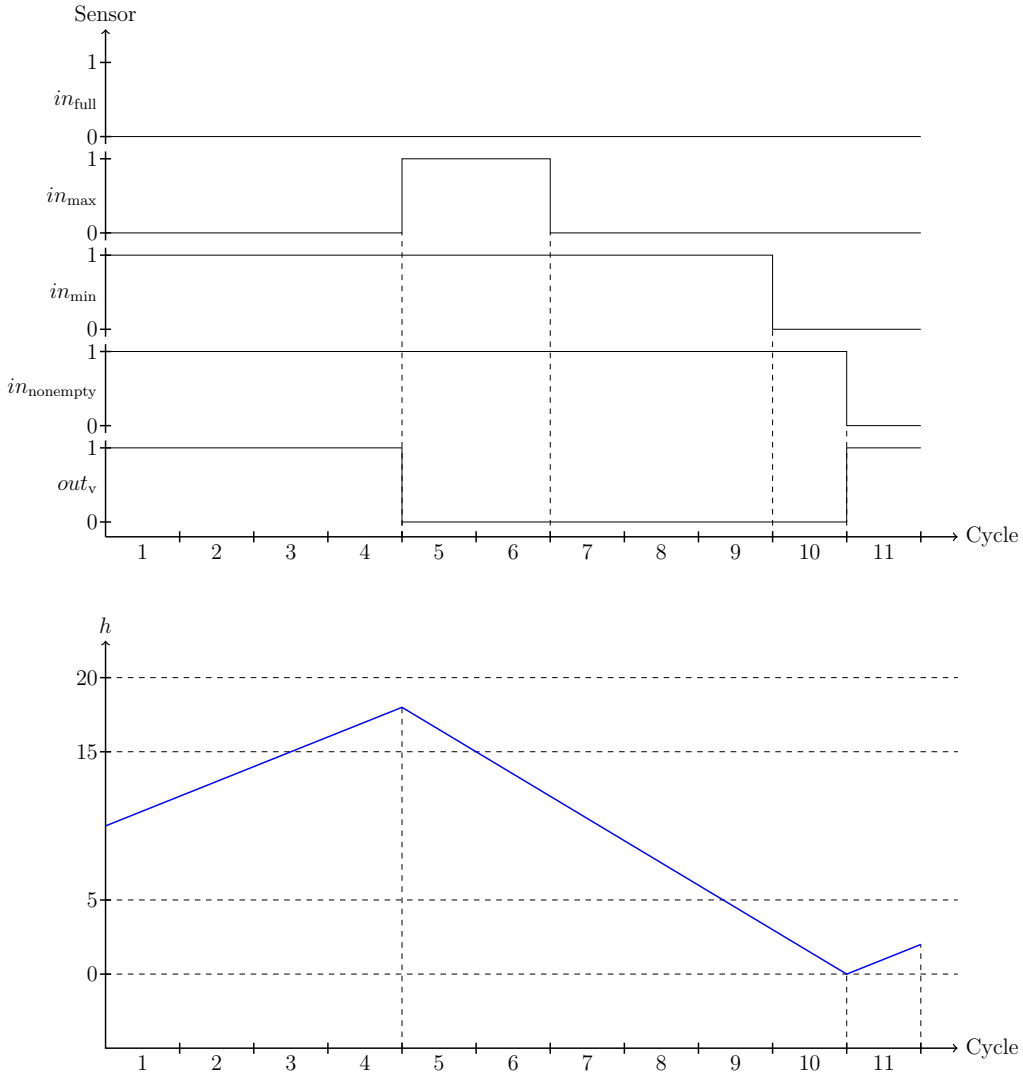

Figure 6.7: *Discrete And Dynamic Behaviour*

As Figure 6.8 shows, the over-approximation causes h to assume values smaller than 0 in the 10th cycle. This allows the system to reach the last cycle, which should not be possible. In such cases, either the parameters have to be set more precisely or PHAVer can be applied, if the resulting model is a linear hybrid automaton. As shown in Figure 6.9 the exact computation of h performed by the PHAVer scenario, does not reach the last cycle. Thus, the analysis might falsely confirm the counterexample given in Equations (6.37) to (6.41) if the hybrid reachability analysis is too imprecise.

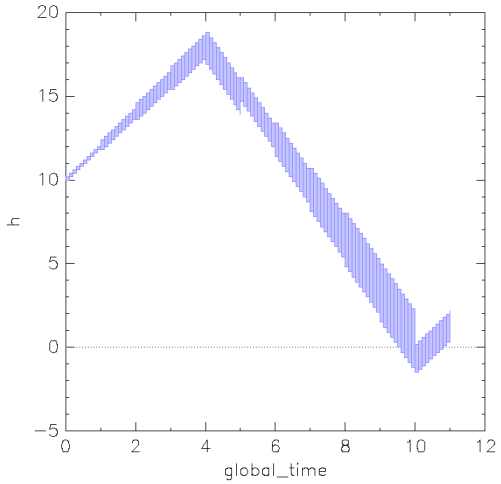

Figure 6.8: *LGG Over-Approximation*

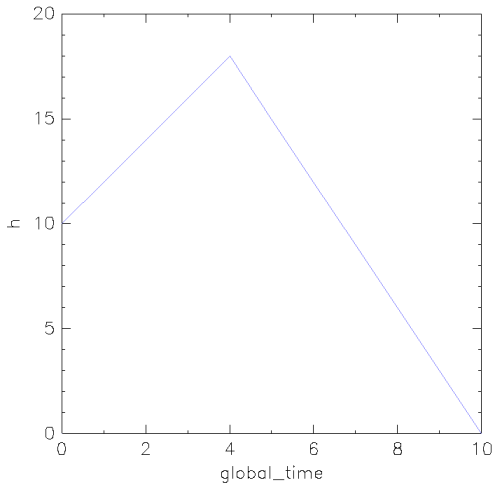

Figure 6.9: *PHAVer Exact Computation*

Tank System Failure

In this section we analyze a faulty control program for the tank system presented in Section 6.1.1. The problem consist of the valve out_{v} not being opened when the water level falls below sensor in_{min}. We still consider the modified conditional ODE systems as presented in Equations (6.35) and (6.36). The problem in the tank system control program allows the tank to run dry.

After 87 iterations, the BMC finds a counterexample of length 5, which can also occur during the hybrid analysis. The counterexample consists of a sequence of the sensors level falling below in_{min} after two cycles and then falling below in_{nonempty} after two more cycles, thus reaching a state where in_{nonempty} is not active. The counterexample is shown in Equations (6.42) to (6.46).

$$in_{\mathrm{full}} : (0,0,0,0,0) \tag{6.42}$$
$$in_{\mathrm{max}} : (0,0,0,0,0) \tag{6.43}$$
$$in_{\mathrm{min}} : (1,1,0,0,0) \tag{6.44}$$
$$in_{\mathrm{nonempty}} : (1,1,1,1,0) \tag{6.45}$$
$$out_{\mathrm{v}} : (0,0,0,0,0) \tag{6.46}$$

Comparing the discrete behaviour with the precise dynamic behaviour allows us to show, that the counterexample is also a counterexample, which can actually occur in the hybrid system. Thus, the counterexample is a hybrid counterexample for the tank system. The comparison of the discrete behaviour and plant dynamics is illustrated in Figure 6.10.

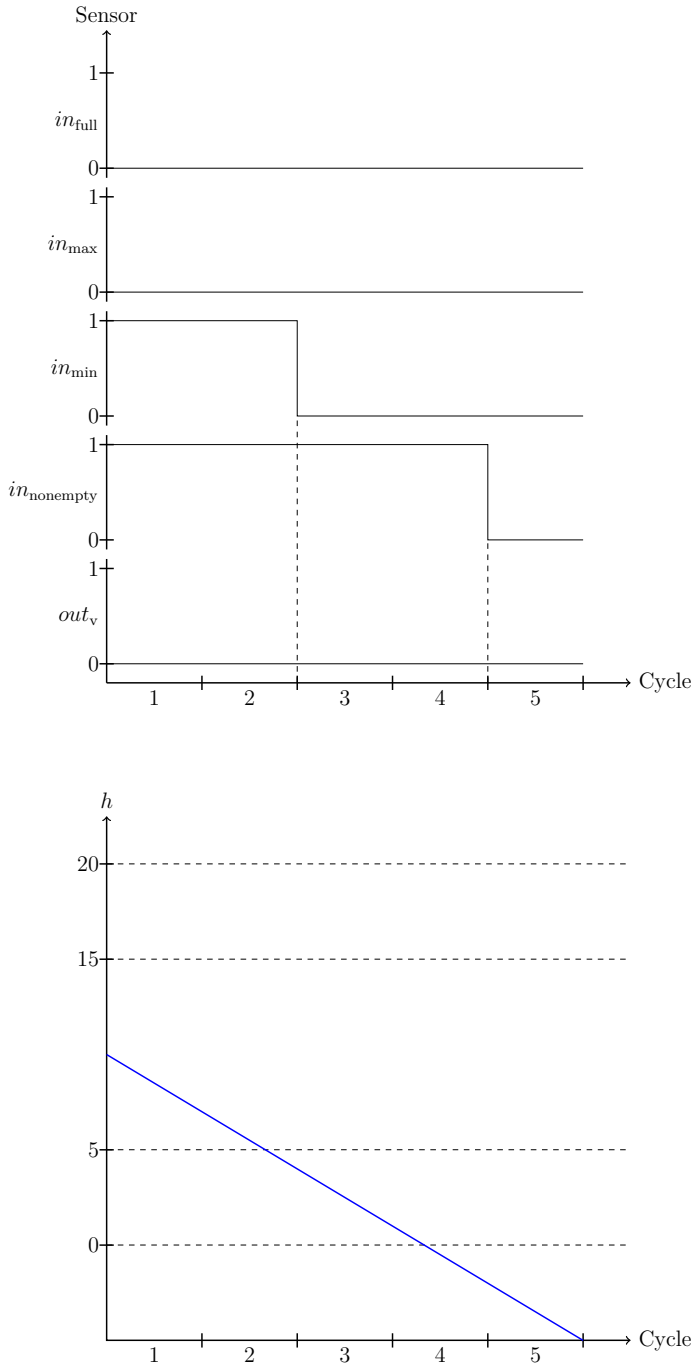

Figure 6.10: *Discrete And Dynamic Behaviour*

The value of h falls below 0 in cycle 4, thus the sensor assignment of $\neg in_{\text{nonempty}}$ corresponds to the dynamic behaviour. We can not generalize that if a counterexample is confirmed by the hybrid automaton, that the system is unsafe, as the unsafe states may only be reachable due to over-approximation.

Considering the analysis with adapted conditional initial values as presented in Equations (6.30) to (6.34), the analysis requires only 81 iterations and finds a possible counterexample with length 4 as shown in Equations (6.47) to (6.51).

$$in_{\text{full}} : (0, 0, 0, 0) \tag{6.47}$$
$$in_{\text{max}} : (0, 0, 0, 0) \tag{6.48}$$
$$in_{\text{min}} : (1, 0, 0, 0) \tag{6.49}$$
$$in_{\text{nonempty}} : (1, 1, 1, 0) \tag{6.50}$$
$$out_{\text{v}} : (0, 0, 0, 0) \tag{6.51}$$

More shorter counterexamples are checked due to the increase of reachable states in the hybrid analysis. Due to h being initialized with $[5, 15)$, a shorter counterexample is found, where $h = 5$ and the system only requires 4 cycles to reach values below 0.

6.2.2 Train System Analysis

In this section we discuss the analysis of the train crossing example. In Section 6.2.2 we show why Zeno behaviour becomes a problem in the hybrid automata generated from the counterexamples of the train crossing example. Furthermore, we provide an analysis of all cycle sequences up to length 8 in Section 6.2.2. In addition, we show an exemplary counterexample of this analysis and the comparison of the discrete and dynamic behaviour.

Train System Zeno Behaviour

The analysis of counterexamples of the train crossing revealed a problem with the current configurations of the verification tool. Zeno behaviour can occur due to dynamic behaviour of the gates appearing in the conditional ODE system conditions. There are two cases, where such behaviour is a problem for a train crossing counterexample. These cases are when the gates are fully closed $g = 0$ or completely opened $g = 90$ in a PLC cycle and the train is in the area 00 or 10 respectively. Thus, either both conditional ODE systems for 00 in Equations (6.12) and (6.13) or the conditional ODE systems for 10 in Equations (6.9) and (6.10) are satisfied allowing the hybrid reachability analysis

to switch between the corresponding locations after each time step in a PLC cycle using the *copy transitions*, which connect the locations.

This is possible as $g = 0$ and $g = 90$ with the areas 00 or 10 satisfy the intersection of the conditions of the conditional ODE systems and the value of g cannot change due to the invariants provided by the conditions of the conditional ODE systems and defined flows. Furthermore, the area cannot change in the PLC cycle as it is defined by discrete variables, which can only change by reaching a successor cycle. Figure 6.11 represents an excerpt of a counterexample automaton, where the gate is fully closed, i.e. $g = 0$, in a PLC cycle i and the train is in the area 00.

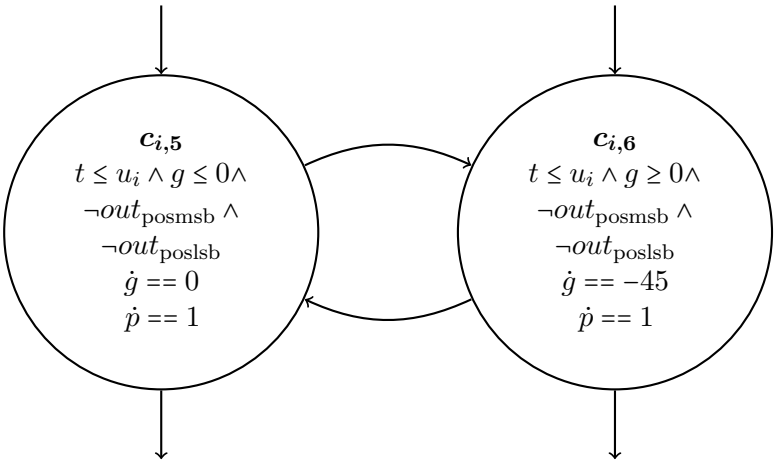

Figure 6.11: *Closed Gates ($g = 0$) at 00 ($out_{posmsb} = 0$ and $out_{poslsb} = 0$)*

If during the execution of the cycle i the area is 00 and $g = 0$ the hybrid reachability can switch between location $c_{i,5}$ and $c_{i,6}$ after each time elapse while $t \leq u_i$. With our current configuration for the verification tool such Zeno behaviour leads to extreme over-approximations, thus many of the computed reachable states are not actually reachable in the system. This behaviour becomes a problem for the counterexample given in Equations (6.52) to (6.58).

$$in_{\text{near}} : (0, 1, 1, 1, 0) \tag{6.52}$$
$$in_{\text{close}} : (0, 0, 0, 0, 0) \tag{6.53}$$
$$in_{\text{far}} : (0, 0, 0, 0, 0) \tag{6.54}$$
$$in_{\text{reset}} : (1, 0, 0, 0, 1) \tag{6.55}$$
$$in_{\text{open}} : (1, 1, 1, 0, 0) \tag{6.56}$$
$$out_{\text{posmsb}} : (1, 0, 0, 0, 0) \tag{6.57}$$
$$out_{\text{poslsb}} : (1, 0, 0, 0, 1) \tag{6.58}$$

The guard $p > 10$ is added, as $in_{\text{reset}} = 1$, to the transitions leading to the last cycle locations should not be satisfiable, as the p evolves constantly with $\dot{p} == 1$, thus should be exactly 4 after the execution of 4 PLC cycles with initial assignment $p = 0$. However, the Zeno behaviour causes the hybrid reachability analysis to over-approximate p as shown in Figure 6.12 and Figure 6.13.

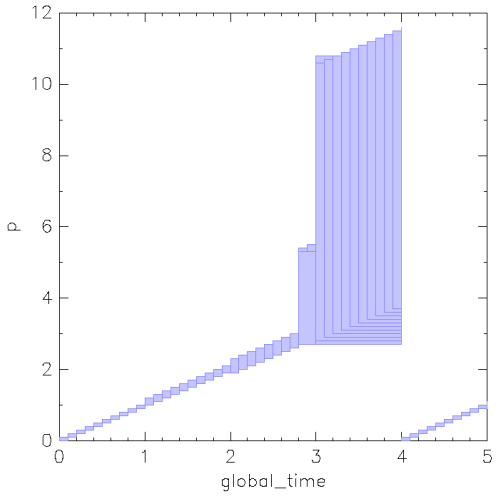

Figure 6.12: *Evolution of p - LGG, Fixed Time Step 0.1, 45 iterations*

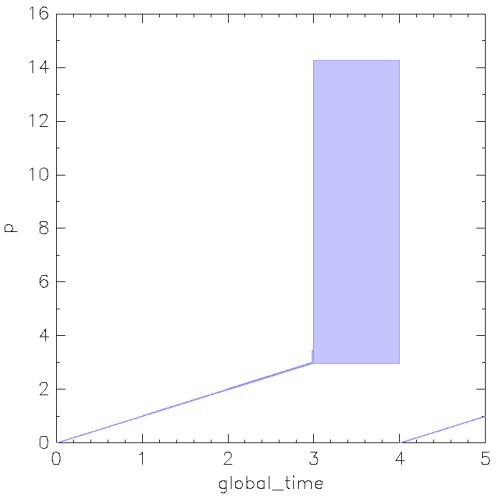

Figure 6.13: *Evolution of p - LGG, Fixed Time Step 0.01, 90 iterations*

Thus it is possible for the train to jump to the in_{reset} sensor without passing in_{close} and in_{far}, which should not be able to happen. Unfortunately, reducing the step size, does not fix the impreciseness resulting from the Zeno behaviour. Furthermore, we consider different bounds on the iterations in all following SpaceEx executions as we require more iterations if the parameters change. In order to verify models exhibiting Zeno behaviour, we have to employ other approaches to reduce the over-approximation.

If we consider a counterexample of an entire round trip of the train, we have exactly two occurrences of this behaviour. The first occurrence is when the gates are completely closed after entering the area 00 and the second occurrence is once the gates are fully opened again while the train is in area 10. A counterexample describing a round trip is given in Equations (6.59) to (6.65).

$$in_{\text{near}} : (0,1,1,1,0,0,0,0,0,0,0) \tag{6.59}$$
$$in_{\text{close}} : (0,0,0,0,1,1,0,0,0,0,0) \tag{6.60}$$
$$in_{\text{far}} : (0,0,0,0,0,0,1,1,1,1,0) \tag{6.61}$$
$$in_{\text{reset}} : (1,0,0,0,0,0,0,0,0,0,1) \tag{6.62}$$
$$in_{\text{open}} : (1,1,1,0,0,0,0,1,1,1,1) \tag{6.63}$$
$$out_{\text{posmsb}} : (1,0,0,0,0,0,1,1,1,1,1) \tag{6.64}$$
$$out_{\text{poslsb}} : (1,0,0,0,1,1,0,0,0,0,1) \tag{6.65}$$

The train starts out at in_{reset} and reaches the in_{near} in the next PLC cycle. The train is then in 00, where Zeno behaviour occurs due to the gates closing and being completely closed as described previously. Afterwards, the train passes sensors in_{close} and in_{far}. Once the train is in area 10, the second case of Zeno behaviour occurs in the constructed counterexample automaton. This Zeno behaviour leads to a heavy over-approximation of the train position p. Figure 6.14 and Figure 6.15 show the reachable values for p and g.

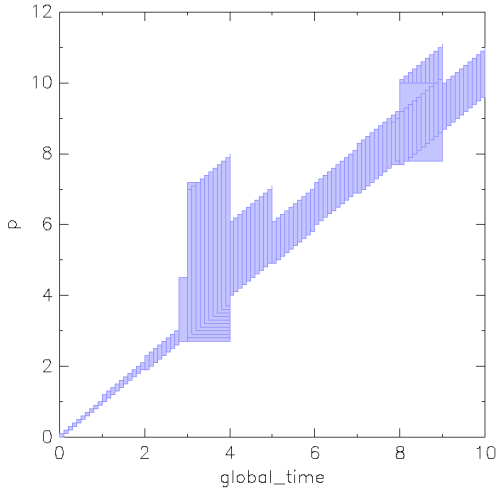

Figure 6.14: *Evolution of p - LGG, Fixed Time Step 0.1, 45 iterations*

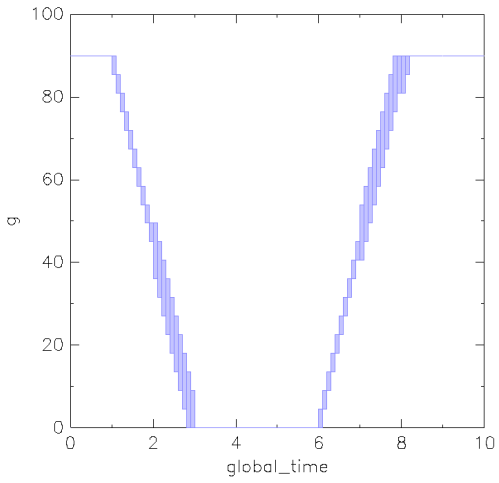

Figure 6.15: *Evolution of g - LGG, Fixed Time Step 0.1, 45 iterations*

As we can see, during cycle 3 and 4 where the gate is fully closed $g = 0$ and cycle 8 where the gate is completely opened $g = 90$, the value of p is heavily over-approximated due to the occuring Zeno behaviour. SpaceEx offers us some configurations to compute more precise approximations of the convex hulls which are used to represent the reachable values of p.

Firstly, we can set the *clustering* parameter, which iteratively replaces a group of sets of a flowpipe with a single convex set. The parameter is defined as a percentage, where 0 percent means that no groups are clustered and at a 100 percent all groups are clustered into a single set. As the clustering step creates a certain number of convex sets, each of these sets results in a new flowpipe during the next time elapse. This might increase the number of iterations needed to reach the last PLC cycle location immensely, but provides a more precise approximation of the reachable states. The default value for the *clustering* is 30. The *aggregation* setting determines whether SpaceEx over-approximates these sets by constructing their convex hull. Configuring these parameters appropriately, we can avoid the heavy over-approximation of p with time step size 0.01 as seen in Figure 6.16 and Figure 6.17.

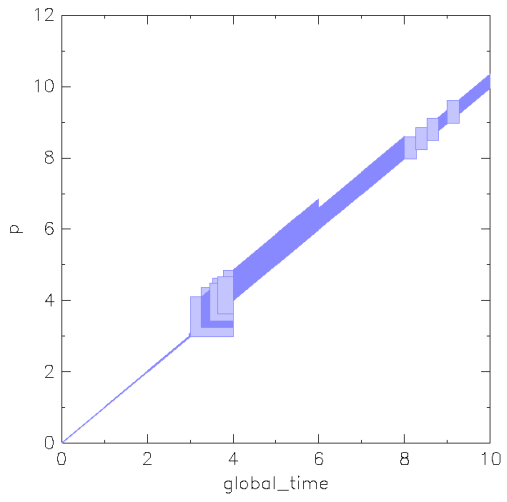

Figure 6.16: *Evolution of p - LGG, Fixed Time Step 0.01, 90 iterations (no aggregation, clustering* 25*)*

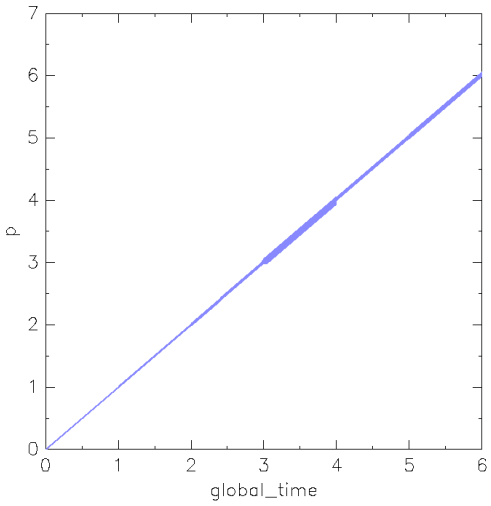

Figure 6.17: *Evolution of p - LGG, Fixed Time Step 0.01, 150 iterations (no aggregation, clustering* 0*)*

The analysis with clustering 25 and no set *aggregation* computes a more precise result, but is still inaccurate as seen in Figure 6.16. In Figure 6.17, the analysis does not even reach the cycles after 6 as the iterations are set too low. Computing more iterations for a *clustering* of 0 and no *aggregation*, results in extremely long computation times. Thus, defining a *clustering* of 0 while the *aggregation* is turned off computes a precise set of reachable states for the analysis, but is not a sensible solution due to the high computational time as shown in Table 6.2. For the analysis to be meaningful, we have determined a compromise for the *clustering* at five percent, while the *aggregation* is turned off and performing 50 iterations for counterexamples up to length 10.

Another possibility to fix the Zeno behaviour problem is to improve the counterexample automaton. If we consider the conditional ODE system (Equation (6.12)) describing the gates closing while the train is in area 00 and the conditional ODE system (Equation (6.13)) as the state where the gates are closed and stay closed in area 00, the system should not be able to switch back from the location where the movement of the gates has stopped to the location where the gates are closing. As once the gates are fully closed, the system should no longer be trying to close the gates, we try to remove this possibility. If we remove the *copy transitions*, which allow this switch back to the closing gate location, the Zeno behaviour is no longer a problem. Analogoulsy, this can be done for the case of a fully opened gate $g = 90$. The results of the SpaceEx analysis for this new model are shown in Figure 6.18 and Figure 6.19 with *clustering* set to default and set *aggregation* turned on.

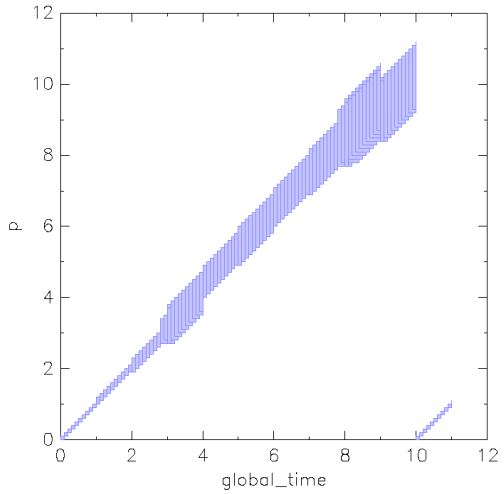

Figure 6.18: *Evolution of p in the Modified Model - LGG, Fixed Time Step 0.1, 45 iterations*

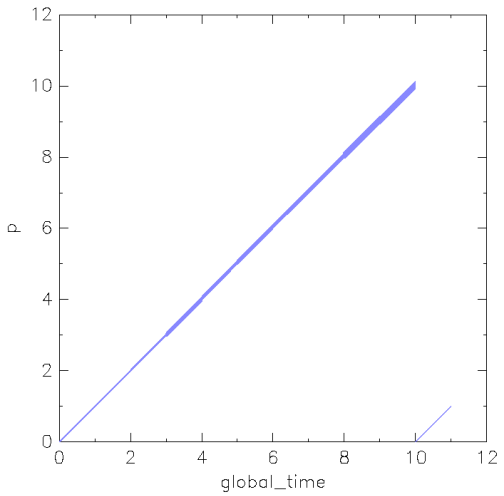

Figure 6.19: *Evolution of p in the Modified Model - LGG, Fixed Time Step 0.01, 45 iterations*

The original time step size 0.1 for the modified model still produces imprecise results as shown in Figure 6.18. Computing the hybrid reachability for a smaller time step size 0.01, provides us with a good result of the reachable values of p.

Another possibility is to use the PHAVer scenario since the over-approximation due to the Zeno behaviour does not occur, as PHAVer computes the precise values for p and g. This solution is applicable as the automaton resulting from the train crossing counterexample is a linear hybrid automaton. In systems with non-linear dynamics, PHAVer can not be applied. As there is no more Zeno behavior during the reachability analysis, we do not need to restrict the

number of iterations. Figure 6.20 and Figure 6.21 shows the computation of p and g for the PHAVer scenario on the original model.

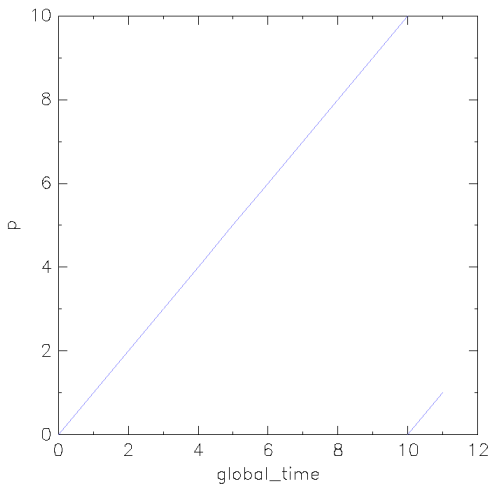

Figure 6.20: *Evolution of p - PHAVer*

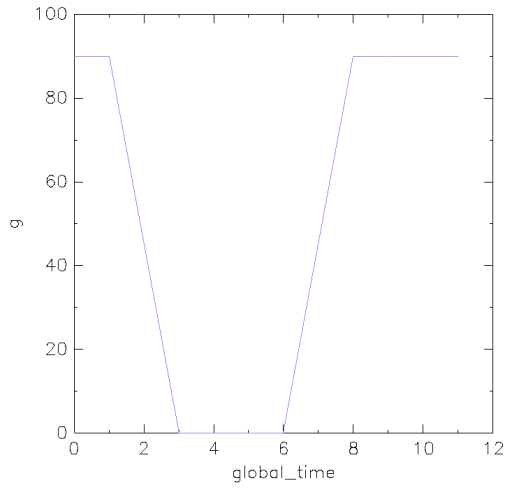

Figure 6.21: *Evolution of g - PHAVer*

Reconfiguring SpaceEx is the easiest solution, but increases the computational time of SpaceEx. The modification of the model is the most time efficient solution, but requires information about which *copy transitions* should be omitted and are not applicable if the system should be able to switch back to the original location. Table 6.2 shows the runtimes of the original heavily over-approximating analysis, the analysis with modified parameters and the analysis of the modified model.

Component	Time Step	Iterations	Runtime (s)
Original Model	0.1	45	2.580
Original Model	0.01	90	22.010
Modified Parameters (25)	0.01	90	14.230
Modified Parameters (0)	0.01	150	153.580
Modified Model	0.1	45	1.920
Modified Model	0.01	45	4.840
PHAVer - Original Model	–	–	0.064

Table 6.2: *The runtime is given with millisecond accuracy for the* Original Model *and parameters, the model with* Modified Parameters *for* clustering 0 *and* 25 *and the* Modified Model. PHAVer *considers the analysis of the original model using the SpaceEx PHAVer scenario*

Train System 8 Cycles

In this section we analyze 8 cycles of the train crossing example. Due to the occurring Zeno behaviour as described in Section 6.2.2 we use the LGG scenario with appropriate parameters. The SpaceEx iterations can be restricted to 50 with no set *aggregation* and clustering at 5. These parameters allow us to perform an efficient analysis, which is precise enough to yield meaningful results. The current control program of the train crossing makes assumptions about the settings of sensors in_{near}, in_{close}, in_{far} and in_{reset}. Thus, during each PLC cycle the program assumes, that only a single sensor may be active at any given time.

The analysis requires 194 to analyze the system with 8 cycles. Similar to the tank example, the memory usage of the BMC is a problem causing the test system to run out of memory. In the following we show, how the discrete and dynamic behaviour are used to refute counterexamples. We consider the discrete counterexample of length 8 in Equations (6.66) to (6.72), which is generated during the analysis.

$$in_{\text{near}} : (0,1,1,1,0,0,0,0) \tag{6.66}$$
$$in_{\text{close}} : (0,0,0,0,1,1,0,0) \tag{6.67}$$
$$in_{\text{far}} : (0,0,0,0,0,0,1,0) \tag{6.68}$$
$$in_{\text{reset}} : (1,0,0,0,0,0,0,1) \tag{6.69}$$
$$in_{\text{open}} : (1,1,1,1,0,0,0,0) \tag{6.70}$$
$$out_{\text{posmsb}} : (1,1,0,0,0,0,0,1) \tag{6.71}$$
$$out_{\text{poslsb}} : (1,1,0,0,0,1,1,0) \tag{6.72}$$

Equations (6.66) to (6.72) is a counterexample as the train has passed the in_{reset} sensor while the gates are still closed ($\neg in_{\text{open}}$). We now compare the discrete behaviour with the dynamic behaviour of p and g in Figure 6.22.

The analysis reveals, that cycle 8 is not reachable, as $p = 7$ at the end of cycle 7 and the guard of the incoming edges of the last location group contains the guard $p \geq 10$ of the replacement rule $in_{\text{reset}} \Leftrightarrow p \geq 10 \Leftrightarrow p := 0$ as marked by the red dots. Thus the last cycle is not reachable in the hybrid system. The generated explanation has length 7 and is shown in Equations (6.73) to (6.79).

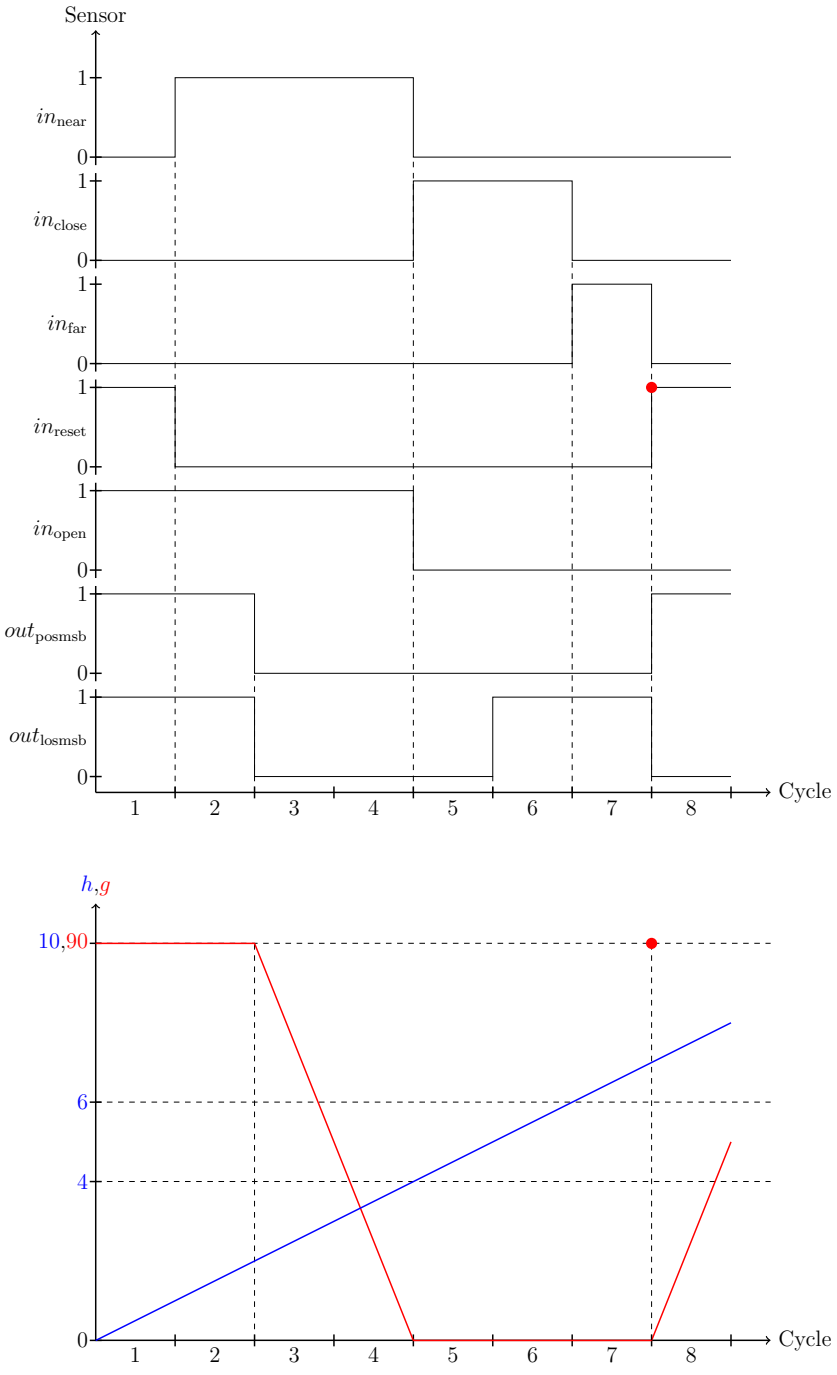

Figure 6.22: *Discrete And Dynamic Behaviour*

$$in_{\text{near}} : (0,1,1,1,0,0,0) \tag{6.73}$$
$$in_{\text{close}} : (0,0,0,0,1,1,0) \tag{6.74}$$
$$in_{\text{far}} : (0,0,0,0,0,0,1) \tag{6.75}$$
$$in_{\text{reset}} : (1,0,0,0,0,0,0) \tag{6.76}$$
$$in_{\text{open}} : (1,1,1,1,0,0,0) \tag{6.77}$$
$$out_{\text{posmsb}} : (1,1,0,0,0,0,0) \tag{6.78}$$
$$out_{\text{poslsb}} : (1,1,0,0,0,1,1) \tag{6.79}$$

As mentioned before, there are currently some issues with the memory usage of the BMC analysis. Thus, the analysis has a similar build up in runtime and memory as the tank example presented in Section 6.3.1.

6.3 Runtime Analysis

After verifying different models for the tank system and the train crossing example as presented in Section 6.1.1 and Section 6.1.2, we evaluate the recorded runtimes. The resulting runtimes allow us to see, which components require the most time. Thus, the runtime analysis allows us to determine, which components can increase the speed of the analysis the most if they are improved. All measurements have been recorded using the system specified in Table 6.3

Processor	Intel(R) Core(TM) i7-2600k, 4x 3.40GHz
Memory (RAM)	16 GB
Operating System	VMWare - Virtual Machine Ubuntu 14.04 LTS (64-Bit)

Table 6.3: *System Specifications*

During the execution of the analysis, all console outputs of the BMC and the hybrid analysis except for an iteration counter have been disabled. All measurements have been taken with millisecond accuracy and the average values have been rounded up to the next millisecond. The total time describes the overall time required by each component considering all iterations. The average time is an average time for each component in a single iteration. Furthermore, min and max time provide us with shortest and longest runtime

of each component considering one iteration. In Section 6.3.1, we present the runtimes resulting from different tank system analyses. In addition, we provide runtime analyses for the train crossing example in Section 6.3.2.

6.3.1 Tank System Runtime

In this section we consider the runtime of the analysis of the tank system. Firstly, the analysis of 10 PLC cycles is shown and discussed. We present how the runtime and the memory usage of the analysis behaves with increasing counterexample length in Section 6.3.1. Afterwards, we analyze the runtime of the tank system with the faulty control program, i.e. the program which never opens the valve in Section 6.3.1. We also provide the runtime of the automaton modeling the entire tank system in Section 6.3.1 as a comparison. Finally, we show how different cycle lengths affect the hybrid analysis, i.e. the automaton generation, hybrid reachability analysis and explanation generation in Section 6.3.1.

Tank System 10 Cycles Runtime

In this section we consider the runtime of the analysis of 10 PLC cycles of the tank system. The analysis stops after 264 iterations, i.e. executions of the BMC and hybrid analysis. During the verification of the tank example, we have also recorded various runtimes for the different steps during the analysis. These measurements allowed us to determine possible weaknesses, i.e. time expensive code segments, in hybrid analysis. We can use these information to improve the analysis in the future. The runtime of the entire analysis of the tank system for a path length of 10 as presented in Section 6.2.1 is given in Table 6.4.

Component	**Total (s)**	**Average (s)**	**Min (s)**	**Max (s)**
Full Analysis	3414.879	12.935	3.227	61.668
BMC Analysis	2417.496	9.157	0.431	58.227
Hybrid Analysis	997.383	3.276	1.934	3.776

Table 6.4: *Total is the runtime of a component considering all* 264 *iterations, while Average, Min and Max are runtimes for a single iteration. Full Analysis is the overall analysis consisting of the BMC and Hybrid Analysis*

As the Table 6.4 shows, the runtime for the BMC is getting increasingly worse for longer cycle sequences. Initially, the BMC requires less than a second, while during the later iterations it requires more than 60 seconds for a single analysis.

Analyzing the runtime of each iteration in detail, we have detected an significant increase in the runtime once the amount of cycles in the produced counterexamples changes. This increase causes the test machine to run out of memory. Table 6.5 shows the increase in the runtime and memory usage.

Cycles	**Avg. Runtime (s)**	**Avg. Memory (MB)**
2	3.112	15.800
3	3.765	17,447
4	4.127	18.555
5	4.783	21.140
6	5.133	34.334
7	6.278	42.554
8	9.778	57.930
9	19.003	140.990
10	57.336	375.668
11	183.222	1162.240
12	881.344	3596.228

Table 6.5: *Average runtimes and memory usage for the iterations of the analysis have been given considering the length of the counterexamples generated by the BMC.*

As we can see in Table 6.5, the analysis requires increasingly more time and memory if the amount of cycles in the counterexample increases. Due to this memory problem, the generation of BMC examples is currently restricted to 11 cycles as the memory of the test system might run out otherwise.

Table 6.4 shows that the increase results in the BMC analysis, as the hybrid analysis always stays between 1.934 and 3.776 seconds. The runtime of the BMC analysis however ranges from 0.431 to 58.227 seconds. These runtime increases occur once the cycle length increases.

Considering the analysis with the conditional initial values as presented in Equations (6.30) to (6.34), the runtime increases as 2174 iterations have to be performed. The runtime of the hybrid analysis in each iteration is affected by the new initial values. While analyzing longer counterexamples, hybrid reachability analysis requires more time. The total runtime of the analysis is shown in Table 6.6.

The increase in iteration results from more states being reachable during the hybrid reachability analysis, thus less counterexamples being excluded during each iteration. Furthermore, the runtime of the hybrid reachability analysis in each iteration increases due to the increased initial states.

Component	Total (s)	Average (s)	Min (s)	Max (s)
Full Analysis	44985.961	20.693	3.547	65.231
BMC Analysis	35221.418	16.201	0.431	58.657
Hybrid Analysis	9764.543	4.491	2.349	8.086

Table 6.6: *Total is the runtime of a component considering all* 2174 *iterations, while Average, Min and Max are runtimes for a single iteration. Full Analysis is the overall analysis consisting of the BMC and Hybrid Analysis with modified initial values*

Tank System Failure Runtime

In this section we consider the runtime of the tank system, which fails due to the valve not opening. The table depicted in Table 6.7 shows us the different runtimes for several components of the analysis, which finished after 87 iterations, i.e. 87 BMC and hybrid analysis executions.

Component	Total (s)	Average (s)	Min (s)	Max (s)
Full Analysis	265.889	3.056	2.772	4.775
BMC Analysis	35.420	0.407	0.357	0.664
Hybrid Analysis	230.469	2.649	2.157	3.273

Table 6.7: *Total is the runtime of a component considering all* 87 *iterations, while Average, Min and Max are runtimes for a single iteration. Full Analysis is the overall analysis consisting of the BMC and Hybrid Analysis*

The times required for the iterations consisting of a BMC analysis and the hybrid analysis are described in the component *Full Analysis*. The time it requires the BMC to determine a counterexample is given in *BMC Analysis*. The times for the component *Hybrid Analysis* are the recorded times of all executions of our hybrid approach, omitting the computation time of the all BMC verifications. If omit the time the SFC Verification requires to initialize the analysis by parsing configuration files, the runtimes for the major hybrid analysis components are shown in Table 6.8.

Component	Total (s)	Average (s)	Min (s)	Max (s)
Automaton Generation	136.841	1.573	1.238	2.051
Reachability Analysis	26.970	0.310	0.188	0.567
Generate Explanation	0.396	0.004	0.003	0.016

Table 6.8: *Total is the runtime of a component considering all* 87 *iterations, while Average, Min and Max are runtimes for a single iteration. The presented components are the major hybrid analysis steps*

The three components describe the *Automaton Generation* as presented in Chapter 3 the *Reachability Analysis* performed by SpacexEx and the *Explanation Generation* as introduced in Chapter 5. As the table shows us, the *Automaton Generation* requires the most time. Thus, we split the generation into smaller components as shown in Table 6.9.

Component	Total (s)	Average (s)	Min (s)	Max (s)
Convert Sequence	39.304	0.451	0.324	0.809
Apply Tool Chain	64.753	0.745	0.643	0.925
Write SpaceEx Model	29.530	0.339	0.180	0.433

Table 6.9: *Total is the runtime of a component considering all* 87 *iterations, while Average, Min and Max are runtimes for a single iteration. The presented components are the model generation steps for the hybrid model*

Convert Sequence is the transformation of the BMC counterexample into a PLC cycle automaton as described in Section 3.1, the extension with PLC cycle times and the application of replacement rules. Furthermore, *Apply Tool Chain* is the execution of the toolchain presented in Section 3.7 and *Write SpaceEx Model* is the component which writes the model files for SpaceEx. As we can see, all three components have high runtimes for small counterexamples. The time of *Write SpaceEx Model* depends on the size of the model as it uses a XML writer to write the model of the XML. A possible solution to improve the time of this component is to reduce the number of transitions and locations, thus reducing the amount of automaton components the writer has to write. We break down the *Convert Sequence* and *Apply Tool Chain* component further in Table 6.10 and Table 6.11.

Component	Total (s)	Average (s)	Min (s)	Max (s)
Initial States	0.174	0.002	0.001	0.006
New Location	0.402	0.001	< 0.001	0.012
Construct Interpretation	0.006	< 0.001	< 0.001	0.001
Dynamic Condition Check	37.647	0.209	0.087	0.790
Add Transitions	0.054	< 0.001	< 0.001	0.001

Table 6.10: *Total is the runtime of a component considering all* 87 *iterations, while Average, Min and Max are runtimes for a single iteration. The presented components the stops of the conversion of a counterexample into a PLC cycle automaton*

The component which requires most of the time is the *Dynamic Condition Check*, which is the SMTInterpol satisfiability check of the replacement rule

conditions. Currently, we transform the logic formulas, which are a data structure provided by the SFC Verification architecture, into conjunctive normal form to be able to parse them more easily before asserting the terms using the SMTInterpol API. We might be able to improve the transformation by parsing the formulas directly without transformation, thus omitting the transformation into conjunctive normal form.

Component	Total (s)	Average (s)	Min (s)	Max (s)
Add Dynamic Behavior	9.269	0.106	0.053	0.153
Add Copy Transitions	0.022	0.003	0.001	0.003
Single Initial Location	42.400	0.487	0.455	0.716
Transform Assignments	0.062	< 0.001	< 0.001	0.008
Guard Disjunction Split	6.924	0.079	0.045	0.155
Invariant Disjunction Split	3.345	0.038	0.020	0.067
Delete Unreachable	0.807	0.009	0.004	0.018

Table 6.11: *Total is the runtime of a component considering all* 87 *iterations, while Average, Min and Max are runtimes for a single iteration. The presented components are the different tools executed in the toolchain*

The broken down runtime analysis of the toolchain reveals, that the construction of the single initial location (SIL) and the conditional initial values in *Single Initial Location* is the problem. On further analysis, we have discovered that the problem is the negation generation of the predecessor conditions as described in Section 3.7.3. However, we can achieve a big performance boost by defining the conditions of the conditional initial values with their predecessor negations and disabling the automatic generation. The comparison of the two approaches is shown in Table 6.12.

Component	Total (s)	Average (s)	Min (s)	Max (s)
SIL Tool	42.400	0.487	0.455	0.716
SIL Tool (no Negation)	7.663	0.089	0.012	0.317

Table 6.12: *Total is the runtime of a component considering all* 87 *iterations, while Average, Min and Max are runtimes for a single iteration. SIL Tool and SIL Tool (no Negation) are the tool to construct the single initial location for the the conditional initial value with and without generating the predecessor negation*

Considering the runtime of the analysis with modified initial values as presented in Equations (6.30) to (6.34), the runtime of the analysis is higher even though less and shorter counterexamples are analyzed since the hybrid analysis requires

more time due to the increase in reachable states. The runtime for the failed analysis with the modified initial values is shown in Table 6.13.

Component	Total (s)	Average (s)	Min (s)	Max (s)
Full Analysis	295.327	3.656	2.472	5.104
BMC Analysis	40.420	0.499	0.311	0.588
Hybrid Analysis	254.907	3.147	2.157	4.273

Table 6.13: *Total is the runtime of a component considering all* 81 *iterations, while Average, Min and Max are runtimes for a single iteration. Full Analysis is the overall analysis consisting of the BMC and Hybrid Analysis with modified initial values*

Tank System Full Automaton Runtimes

In this section we consider a hybrid automaton modeling the entire tank system, i.e. its discrete and dynamic behaviour. Unfortunately, for small systems like the tank system it is more efficient to model the entire composition of discrete and dynamic behaviour. A model for the tank system has been presented in Section 6.1.1. As we only require a single hybrid reachability analysis to verify the system for a specific amount of PLC cycles. As there are two initial states and only two paths which can be taken, verifying n cycle executions requires $2n$ iterations. Figure 6.23 and Figure 6.24 shows the execution of 24 iterations for the PHAver and LGG Support Function scenario.

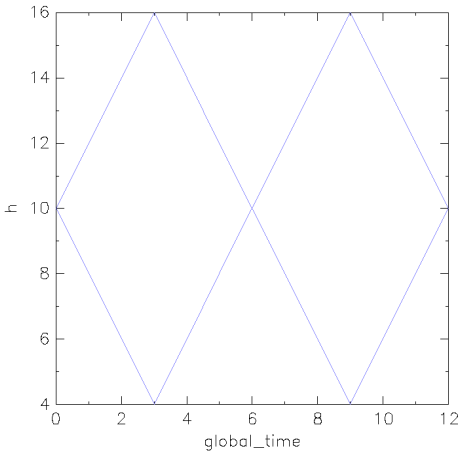

Figure 6.23: *PHAVer - Full System*

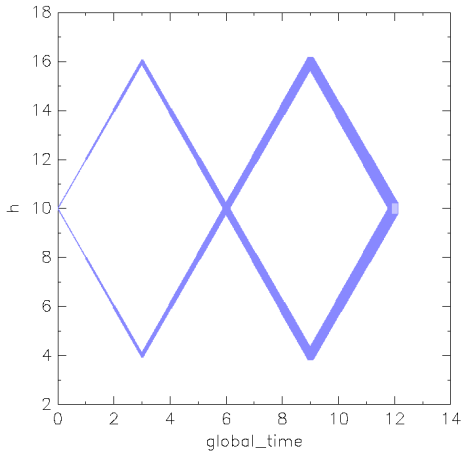

Figure 6.24: *LGG - Full System*

Comparing the runtime of verifying the hybrid automaton with the approach presented in this thesis shows, that for the tank system, constructing a hybrid automaton as proposed in Section 6.1.1 is the better solution to verify the system. The runtime of the analysis for different cycle lengths and the PHAVer and LGG scenario is shown in Table 6.14.

Cycles / **Scenario**	**10 (s)**	**50 (s)**	**100 (s)**	**500 (s)**	**1000 (s)**
PHAVer	0.10	0.13	0.20	1.53	5.04
LGG (0.1)	0.21	0.96	2.70	47.15	178.13
LGG (0.01)	0.64	2.33	5.84	59.82	212.42

Table 6.14: *The runtimes presented are for a specific amount of PLC cycles and SpaceEx Scenarios. For the LGG scenario, we use the fixed time step sizes of* 0.1 *and* 0.01.

Each hybrid reachability analysis verifies the given system as no forbidden states are reached. The values of h stay in $2 \leq h \leq 18$ even when the reachable values are over-approximated. The runtimes of the verifications are currently still distinctly better than of the BMC analysis as presented in Section 6.3.1.

Tank System Counterexample Runtimes

In addition to the BMC analysis, we have performed a runtime analysis on counterexample sequences with different lengths for the tank example. The analyzed sequence lengths are 10, 50, 100, 500 and 1000. The measurements represent averages over 10 different counterexamples for each sequence length. Furthermore, the sequences are constructed in such a way, that the analysis reaches the last cycles. Table 6.15 shows the measured times for the tank system counterexamples for length 10, 50, 100, 500 and 1000.

Table 6.15 shows, that the runtime of the automaton construction, explanation generation as well as the hybrid reachability analysis approximately increases linearly. Considering the constructed hybrid automata, we show how the cycle length affects the number of automaton components in Table 6.16.

The components increase linearly with increasing amount of PLC cycles. We also consider the system without *copy transitions* as the conditions of the conditional ODE systems only contain discrete variables, thus allowing us to omit the *copy transitions* between the conditional ODE locations.

Component \ Cycles	**10 (s)**	**50 (s)**	**100 (s)**	**500 (s)**	**1000 (s)**
HA Generation					
Convert Sequence	2.763	8.221	12.613	29.847	49.315
Apply Toolchain	1.177	1.885	2.075	9.554	12.215
SpaceEx Model Writer	1.862	3.559	6.230	16.202	28.014
SpaceEx Analysis					
LGG (0.1)	0.211	0.955	1.883	8.476	18.233
LGG (0.01)	2.387	13.101	23.341	115.432	283.114
PHAVer	0.101	0.305	0.477	2.664	5.246
Explanations					
Generate Explanation	0.047	0.071	0.108	0.256	0.455

Table 6.15: *The measured runtimes are averages of the computation of* 10 *counterexamples for the tank system of length* 10, 50, 100, 500 *and* 1000, *respectively. The runtimes are given for the different components of the automaton generation as well as the explanation generation. Furthermore, it provides the SpaceEx execution times for the LGG scenario with time step size* 0.1 *and* 0.01 *and the PHAVer scenario.*

Cycles	**Locations**	**Transitions(NC)**	**Variables**
10	21	66(46)	7
50	101	306(206)	7
100	201	606(406)	7
500	1001	3006(2006)	7
1000	2001	6006(4006)	7

Table 6.16: *The table provides information about the number of components in the generated hybrid automaton of the tank system. The number of locations, transitions with the generation of copy transitions (and without the generation of copy transitions) as well as the amount of variables.*

6.3.2 Train System Runtime

In this section, we consider the runtime of the train crossing example as introduced in Section 6.1.2. Firstly, we analyze the analysis of 8 PLC cycles in Section 6.3.2. In addition, we have performed a runtime analysis of the hybrid analysis on several sequences for the train crossing example in Section 6.3.2.

Train System 8 Cycles Runtimes

In this section we analyze the runtime of the verification of 8 PLC cycles of the train crossing. The analysis terminates after 194 iterations, i.e. executions of the BMC and hybrid analysis. The runtime analysis of the verification as presented in Section 6.2.2 is given in Table 6.17.

Component	Total (s)	Average (s)	Min (s)	Max (s)
Full Analysis	11642.315	59.920	4.557	252.844
BMC Analysis	9885.594	50.957	0.752	233.114
Hybrid Analysis	1756.721	9.055	3.063	15.967

Table 6.17: *Total is the runtime of a component considering all* 194 *iterations, while Average, Min and Max are runtimes for a single iteration. Full Analysis is the overall analysis consisting of the BMC and Hybrid Analysis*

As the Table 6.17 shows, that longer cycle sequences require more time in the BMC analysis. At the beginning of the analysis, the BMC requires less than a second to generate a counterexample, while later iterations require more than 4 minutes to find a counterexample.

Train System Counterexample Runtimes

Similar to the runtime analysis of the counterexamples for the tank system provided in Section 6.3.1, we consider counterexamples with different lengths for the train crossing. The analyzed counterexamples have the lengths are 10, 50, 100, 500 and 1000. All measurements are averages over the analysis of 10 train crossing counterexamples for each length, where the last cycle is reachable during the hybrid reachability analysis. Table 6.18 shows the measured times for the train counterexamples.

Component \ Cycles	10 (s)	50 (s)	100 (s)	500 (s)	1000 (s)
HA Generation					
Convert Sequence	1.622	4.252	8.090	22.275	34.335
Apply Toolchain	2.302	2.242	5.197	15.871	35.261
SpaceEx Model Writer	5.622	14.072	23.915	90.647	178.725
SpaceEx Analysis					
PHAVer	0.511	2.231	4.030	23.714	49.113
Explanations					
Generate Explanation	0.052	0.078	0.111	0.293	0.516

Table 6.18: *The measured runtimes are averages of the computation of* 10 *counterexamples for the train crossing of length* 10*,* 50*,* 100*,* 500 *and* 1000*, respectively. The runtimes are given for the different components of the automaton generation as well as the explanation generation. Furthermore, it provides the SpaceEx execution times for the PHAVer scenario.*

We only provide runtimes for the PHAVer scenario as due to the Zeno behaviour occuring in the train example as described in Section 6.2.2, causes the LGG scenario to require many iterations to reach the last cycle. For 10 cycles we require about 50 iterations for the LGG analysis, while 50 cycles already requires about 600 iterations. Similarly to the tank example the runtime for the construction of the automaton increases approximately linearly to the counterexample length. In addition to the runtimes, we provide the amount of hybrid automaton components for the counterexamples in Table 6.19.

Cycles	Locations	Transitions	Variables
10	61	672	10
50	301	3312	10
100	601	6612	10
500	3001	33012	10
1000	6001	66012	10

Table 6.19: *The table provides information about the number of components in the generated hybrid automaton of the train crossing. The number of locations, transitions with the generation of copy transitions as well as the amount of variables.*

The amount of components increases linearly, while the runtime of the automaton and explanation generation approximately increases linearly in respect to the number of PLC cycles contained in the counterexamples.

6.3.3 Improvements

We have several possibilities to improve the hybrid analysis in respect to its runtime and the generation of the hybrid automaton. In this section we discuss approaches to reduce the complexity of the automaton as well as the runtime of the hybrid analysis.

The first possibility is to configure the verification tools SpaceEx and Flow* as presented in Section 2.5.1 and Section 2.5.2 with coarser parameters. This approach has to be used cautiously, as parameters, which are too rough, might lead to meaningless results. For example configuring a iteration count for the discrete and time steps, which is too low to reach the last group of locations in the counterexample automaton, does not verify the counterexamples with dynamic behaviour properly. If the plant dynamics are linear, i.e. the automaton resulting from a counterexample is a linear hybrid automaton, the PHAVer scenario can be used to compute precise results efficiently.

As we have seen in the examples evaluated in Section 6.1.1 and Section 6.1.2, if we define the conditions of the conditional ODE systems so they cover the entire domain of their variables, we can omit the negated condition of all conditional ODEs. The usual construction which adds the conditional ODE systems to the counterexample automaton is presented in Section 3.5. As all other conditions cover the domain, their negation would result in an unsatisfiable formula. Constructing the negated condition for the conditional ODEs seen in Equations (6.8) to (6.13) results an elaborate formula, which is unsatisfiable. We are able to reduce the runtime of the automaton generation by omitting this construction. This can be accomplished by adding a new tag to the XML containing the conditional ODEs as prsented in Section 4.1.1. The new tag is inserted inside the `<condODEsys>` tag as shown in Listing 6.4

```
<?xml version="1.0" encoding="UTF-8"?>
<condODEsys>
    <addNegatedTerms>false</addNegatedTerms>
    ...
</condODEsys>
```

Listing 6.4: *Add Negated Terms*

The new tag `<addNegatedTerms>` allows us to specify whether the negation of the conditional ODEs is going to be added in a new location as described in Section 3.5 (*true*) or if it is going to be omitted (*false*). The default setting for this configuration, if the `<addNegatedTerms>` tag is not set, is *true*.

In certain systems we can also omit the *copy transitions* as explained in Section 3.7.2. If the conditions of the conditional ODE systems only contain discrete

variables in their conditions, we can omit the *copy transitions*, as the values of the discrete variables can only change if a new cycle is reached. The tank example in Section 6.1.1 is such a system, as the conditions of the conditional ODEs only consist of discrete variables in this case out_v. This approach reduces the number of transitions. The number of transitions depends on the number of PLC cycles n and the amount of conditional ODE systems m. Assuming we generate the negation for the conditional ODE systems the number of transitions is $m \cdot (m+1) \cdot n$ as the $(m+1)$th location is generated for the negation. All these locations have m transitions to all other locations in the cycle and there are n cycle locations.

6.4 Summary

In this chapter we presented two different PLC controlled plants. The first system is a tank system, which we define for the BMC analysis and plant dynamics. Moreover, the entire system is modeled as a single hybrid automaton. The train crossing models a train, which passes a street crossing where gates are raised and lowered. The analysis of the tank system contains a full analysis bounded by the amounts of PLC cycles and a comparison of the discrete and dynamic behaviour, a discussion of the problems caused by over-approximation and the analysis which is repeated for a faulty tank system control program. For the train system a similar analysis is performed. The system is analyzed for a specific amount of PLC cycles where the discrete and dynamic behaviour is compared. Furthermore, we recorded the runtimes of these analyses. We evaluate these runtimes to determine weaknesses in the algorithms. Additionally, we determine the runtime of the hybrid automaton representing the entire tank system as a comparison to our new approach and analyze the performance of the hybrid analysis for counterexamples of different lengths for the tank and train example. Afterwards, we introduce possible improvements, which can be used to analyze the systems more efficiently.

Chapter 7

Conclusion and Future Work

The proposed novel approach for the verification of PLC-controlled plants provides an automated verification process. The verification works iteratively by performing a bounded model checking analysis, an automaton construction and a hybrid reachability analysis during each iteration. The initial input control program is analyzed using bounded model checking. Either the bounded model checking ascertains that the model derived from the program is safe or it produces a counterexample and stores the current state of the BMC analysis. This counterexample contains a path consisting of the execution of PLC scan cycles and their discrete input and output variable assignments.

The counterexamples of the BMC are used to construct hybrid automata, reducing the combination of discrete and hybrid behaviour to the paths described by the counterexamples. The plant dynamics defined by conditional ODE systems, conditional initial values and replacement rules are added to this automaton. Afterwards, the model describing the discrete counterexample path as well as the dynamic behaviour of the plant can be verified using a third-party tool as SpaceEx or Flow* to perform a hybrid reachability analysis. If the execution of the discrete path is replicable considering the dynamic behaviour, i.e., a location associated with the last PLC cycle is reached, the result is *unknown* as we cannot falsify a model due to the over-approximation in the hybrid reachability analysis. However, if the dynamic behaviour prohibits the hybrid reachability analysis from reaching any location associated with the last cycle, an explanation is constructed. In this case, the counterexample generated by the BMC is not a counterexample if the plant dynamics are considered. The explanation is then used by the BMC to exclude the given counterexample and counterexamples with the same prefix. Thereafter, the BMC analysis is resumed and the next iteration is started. This process is continued until either the hybrid analysis confirms a BMC counterexample or the system is proven to be safe by the BMC analysis.

The counterexample automaton generation extends the path of executed PLC scan cycles into an automaton representing a possible path which can occur during the discrete analysis performed by the BMC while considering the plant dynamics. Because we do not combine the entire discrete behaviour with the plant dynamics, we avoid the state-space explosion which might arise in larger systems. During the work on the new verification approach we have recognized which parameters and properties affect the runtime of the analysis the most. As each iteration contains the execution of a bounded model checker, the

automaton generation and a hybrid reachability analysis, we try to exclude as many counterexamples as possible during each iteration. Thus, reducing the iterations needed to verify the system. The presented method should be applied on larger models as for smaller models the runtime might be higher compared to the combination of the entire discrete and dynamic behaviour.

The evaluation of the discussed examples has revealed problems concerning the automaton construction. We have shown how to avoid the problems which occur due the creation of logic formula negations while extending the automaton with the conditional ODE systems and conditional initial values. In both cases, we can define the conditions covering the entire domain, avoiding the negation construction. This solution strongly reduces the automaton construction time. Zeno behaviour has proven to be a problem in some models as it causes heavy over-approximations of some variables. We have presented solutions to reduces this over-approximation and to eliminate the Zeno behaviour in certain models.

The runtime of the entire analysis becomes increasingly worse for longer cycle sequences. Unfortunately, the memory usage of the BMC analysis also increases up to the point where the test machine runs out of memory. The required time for the hybrid analysis however does not increase as badly as the runtime of the BMC analysis. During the hybrid reachability analysis, we have also ascertained that with increasing amount of PLC cycles, the parameters of the hybrid reachability analysis have to be adapted to perform more precise verifications. Otherwise, the analysis may over-approximate too much and may reach states, which are actually unreachable. A future goal is to develop heuristics to adapt the parameters according to the system and the number of PLC cycles which need to be analyzed.

In the future, the analysis should provide full support for integer values in the BMC counterexamples as well as integer wildcards. This extension allows to model more complex systems as well as improve upon the represented systems. The position of the train in the train crossing example could be modeled as an integer instead of two boolean variables.

Another major future goal is to provide explanations, which exclude more counterexamples in an iteration. We have already presented some improvements by allowing wildcards in the explanations. However, we still require heuristics to construct appropriate explanations using wildcards. Furthermore, the explanations are currently prefixes of counterexamples. Another improvement would be to allow explanations describing subsequences which should be excluded during the BMC execution.

The exemplary verifications we have conducted showed that in some systems the hybrid reachability analysis might over-approximate heavily due to Zeno

behaviour. The analysis reaches values for the continuous variables which are not reachable in the actual system and may confirm counterexamples with dynamic behaviour which actually does not correspond to the plant dynamics. We have proposed solutions to handle such behaviour. These solutions also contain a transformation of the model, which requires new information to decide how the model is going to be modified. Such a transformation is shown for the train system, where the Zeno behaviour occurs due to the analysis switching between the locations where the gate is staying closed and where it is closing even though the gate is fully closed already. In the transformation, copy transitions between the locations are removed as it is not sensible for the hybrid model to return to the closing state and to try closing an already closed gate.

The runtime analysis of the verification approach has provided information about the weaknesses of the current algorithms. Some of these problems could be solved by improving the used model. As previously mentioned the negation generation can be easily avoided. The runtime analysis of the counterexamples with different lengths showed that the runtime for the hybrid analysis increases linearly according to the counterexample length.

In conclusion, the new approach provides a promising reduction of the size of the hybrid models by combining a discrete and hybrid verification. The hybrid reachability analysis is reduced to a single discrete path which is extended with the plant dynamics in each iteration avoiding a possible state space explosion in larger system. Thus, in the future the approach should allow us to analyze large PLC controlled systems efficiently.

Bibliography

[ACH+95] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, et al. The Algorithmic Analysis of Hybrid Systems. *Theoretical Computer Science*, 138:3–34, 1995.
(cited on pages 2, 13, 16 and 17)

[BCC+03] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded Model Checking. volume 58 of *Advances in Computers*, pages 117 – 148. Elsevier, 2003.
(cited on page 1)

[BCMP98] Luciano Baresi, Stefania Carmeli, Antonello Monti, and Mauro Pezzé:. PLC Programming Languages: A Formal Approach. Associazione Nazionale Italiana Per L'Automazione, 1998.
(cited on pages 1, 6)

[BM98] Martin Berz and Kyoko Makino. Verified Integration of ODEs and Flows Using Differential Algebraic Methods on High-Order Taylor Models. *Reliable Computing*, volume 4:pages 361–369, 1998.
(cited on page 20)

[BST10] Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.
(cited on page 25)

[CAS12] Xin Chen, Erika Abraham, and Sriram Sankaranarayanan. Taylor Model Flowpipe Construction for Non-linear Hybrid Systems. In *Proc. of the 33rd IEEE Real-Time Systems Symposium (RTSS'12)*, pages 183–192. IEEE Computer Society, 2012.
(cited on page 20)

[CAS13] Xin Chen, Erika Ábrahám, and Sriram Sankaranarayanan. Flow*: An Analyzer for Non-linear Hybrid Systems. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, volume 8044 of *Lecture Notes in Computer Science*, pages 258–263. Springer Berlin Heidelberg, 2013.
(cited on pages 2, 3, 17 and 20)

[CHN12] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. SMTInterpol: An Interpolating SMT Solver. In *SPIN*, pages 248–254, 2012.
(cited on pages 3, 25)

[CK03] Edmund Clarke and Daniel Kroening. Hardware Verification Using ANSI-C Programs As a Reference. In *Proceedings of the 2003 Asia and South Pacific Design Automation Conference*, ASP-DAC '03, pages 308–311, New York, NY, USA, 2003. ACM.
(cited on page 10)

[Dij97] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1997.
(cited on page 10)

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.
(cited on page 11)

[ELS05] Sebastian Engell, Sven Lohmann, and Olaf Stursberg. Verification of Embedded Supervisory Controllers Considering Hybrid Plant Dynamics. In *International Journal of Foundations of Computer Science*, volume 15, pages 307–312, 2005.
(cited on page 1)

[FLGD+11] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, et al. SpaceEx: Scalable Verification of Hybrid Systems. In Shaz Qadeer Ganesh Gopalakrishnan, editor, *Proc. 23rd International Conference on Computer Aided Verification (CAV)*, LNCS. Springer, 2011.
(cited on pages 2, 3 and 17)

[Fre05] Goran Frehse. PHAVer: Algorithmic Verification of Hybrid Systems Past HyTech. In Manfred Morari and Lothar Thiele, editors, *Hybrid Systems: Computation and Control*, volume 3414 of *Lecture Notes in Computer Science*, pages 258–273. Springer Berlin Heidelberg, 2005.
(cited on page 17)

[GG09] Colas Guernic and Antoine Girard. Reachability Analysis of Hybrid Systems Using Support Functions. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, pages 540–554, Berlin, Heidelberg, 2009. Springer-Verlag.
(cited on page 18)

[GHN+04] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): Fast Decision Procedures. In

Rajeev Alur and DoronA. Peled, editors, *Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 175–188. Springer Berlin Heidelberg, 2004.
(cited on page 25)

[Gou09] Brian Gough. *GNU Scientific Library Reference Manual - Third Edition*. Network Theory Ltd., 3rd edition, 2009.
(cited on page 20)

[HG98] Z. Doulgeri Hassapis G., I. Kotini. Validation of a SFC Software Specification by Using Hybrid Automata. In *Proc. of INCOM'98, pages 65-70. Pergamon*, 1998.
(cited on page 1)

[IEC07] *An Open Source IEC 61131-3 Integrated Development Environment*, volume 1, 2007.
(cited on page 7)

[JP00] Michael Joswig and Konrad Polthier. JavaView JVX Format. `http://www.eg-models.de/formats/Format_Jvx.html`, 2000.
(cited on page 19)

[Lat02] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.
(cited on page 8)

[MT00] Robert Maier and Nick Tufillaro. Gnu Plotutils. `http://www.gnu.org/software/plotutils/`, 2000.
(cited on page 19)

[NA12] Johanna Nellen and Erika Ábrahàm. Hybrid Sequential Function Charts. In *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV), Kaiserslautern, Germany, March 5-7, 2012*, pages 109–120, 2012.
(cited on pages 23, 24)

[NA14] Johanna Nellen and Erika Abraham. A CEGAR Approach for the Reachability Analysis of PLC-Controlled Chemical Plants. In *Proc. of the 2nd IEEE International Workshop on Formal Methods Integration (FMi'14)*. IEEE Computer Society Press, 2014.
(cited on pages 1, 22)

[Pol06] Konrad Polthier. Javaview. `http://www.javaview.de`, 2006.
(cited on page 19)

[SC10] Olivier Lebeltel Scott Cotton, Goran Frehse. The SpaceEx Modeling Language. http://spaceex.imag.fr/sites/default/files/spaceex_modeling_language_0.pdf, 2010.
(cited on pages 17, 24)