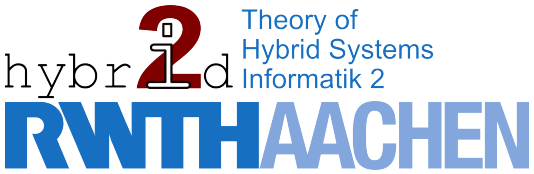

RWTH Aachen University
Rheinisch-Westfälische Technische Hochschule Aachen

Chair of Computer Science 2
Theory of Hybrid Systems
Prof. Dr. Erika Ábrahám

Bachelor Thesis

Counterexample-Guided Abstraction Refinement for Hybrid SFC Verification

Kai Axel Driessen
Matriculation Number: 297607

- September 2012 -

Primary Referee:	Prof. Dr. Erika Ábrahám
Secondary Referee:	Prof. Dr.-Ing. Stefan Kowalewski
Supervisors:	Dipl.-Inform. Johanna Nellen

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 3th of September 2012

Acknowledgments

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

I give thanks to my supervisor Dipl.-Inform. Johanna Nellen, who could always spare some of her time and who gave me helpful advice with me 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 bachelor's course would not have been possible.

Contents

Chapter 1

Introduction

This bachelor thesis deals with the verification of plant controls. The primary goal is to verify chemical plants while keeping their models as simple as possible. The plant behavior is often controlled by programmable logic controllers (PLCs). Users can program PLCs, e.g., by using the graphical programming language of sequential function charts (SFCs).

PLC controlled systems are usually safety-critical. For example, an overflowing tank may cause a system to fail. In general, the verification of plants can be accomplished by analyzing the SFC either in isolation or in combination with a model of the plant [BCMP98, HKD98]. Considering this combined approach, timed or hybrid automaton (HA) can be applied to model the SFC, while hybrid automata specify the plant dynamics. Finally, these automata can be verified by using existing tools for hybrid automata reachability analysis. We combined SFCs and plant dynamics into a single hybrid automaton and analyze it. Additionally, our verification approach uses counterexample-guided abstraction refinement (CEGAR).

The resulting hybrid automata modeling the SFCs are likely to become too large to be analyzed. CEGAR techniques are proposed in [ELS05]. An abstraction of the dynamic behavior of the plant reduces the model but can also cause the model to become too coarse due to missing or incomplete behaviour. We propose a technique to use SFCs and conditional ordinary differential equation (ODE) systems to describe the plant dynamics under specific conditions. We combine them to hybrid sequential function charts (HSFCs) [NA12] that can be converted into hybrid automata and verified accordingly.

In order to keep the model size to a minimum during this verification process, we add parts of the plant dynamics to the model successively. Thus, we achieve refinement by extending the HSFC. During this refinement, the above mentioned conditional ODE systems are attached to the HSFC iteratively. In each iteration a third-party tool is used to verify the given safety properties for the current system. Depending on its result, we refine the system if necessary. In case a reduced model is proven safe, the complete model containing all plant dynamics is also safe. On the other hand, if a reduced model is unsafe, we continue integrating the dynamic behavior until the model is either validated or remains unsafe although all available conditional ODE systems have been included. This CEGAR-based verification is illustrated in Figure 1.1.

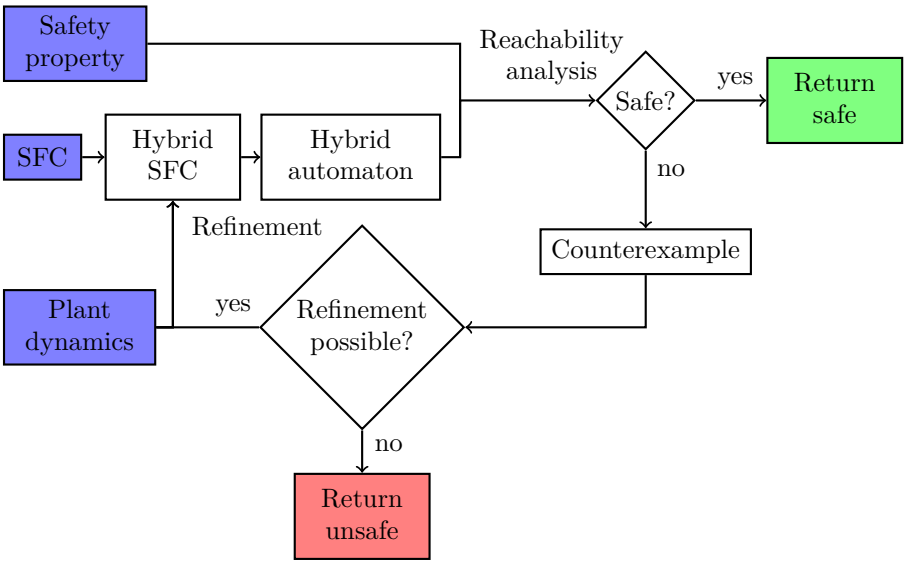

Figure 1.1: *Flow chart of the complete verification process*

With respect to this verification process, some universal conditions are defined determining whether specific plant dynamics should be added to a model. These conditions restrict the size of the resulting hybrid automaton because not all plant dynamics are appropriate for each part of the plant. In order to find the best refinement sequence, we apply different heuristics to analyze the counterexamples provided by an HA verification tool. We present three strategies to iteratively select plant dynamics based on these counterexamples consisting of the output of the HA verification tool.

In order to explain the basic requirements for our novel approach and, subsequently, elaborate on the approach itself, this bachelor thesis is structured in the following way. First, PLCs and HSFCs are discussed in Chapter 2 to present the required data structures and the transformation of HSFCs to hybrid automata. In addition we introduce an exemplary tank system which is referred to in the subsequent chapters as well. The third-party tool SpaceEx [FLGD+11] to analyze hybrid automata is presented. In Chapter 3, the different parts of our CEGAR-based plant verification are described and SpaceEx is integrated into the verification. Instructions to create models are given and the previously introduced tank system is modeled using the third-party Beremiz [Tis12] editor. The counterexamples provided by the verification tool in case of failed verifications are described in detail. The evaluation of these counterexamples to choose appropriate plant dynamics, thus, iteratively extending the current model are described in Chapter 4. Furthermore, we define the conditions under which a system is considered refinable and implement strategies, two naive and one more complex approach, to select the most promising candidates for refinement

during each iteration. In Chapter 5, the tank model and one additional example are evaluated using our verification algorithm, while employing both a naive and a more complex refinement strategy. Chapter 6 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, hybrid sequential function charts are introduced which are diagrams that allow to model both the discrete and the dynamic behavior of a plant. These HSFCs are used as a model in our CEGAR-based plant control verification.

In Section 2.1, an introduction to programmable logic controllers is given as they are used to control chemical plants. The operating principle of PLC scan-cycles is explained. We introduce the syntax and the semantics of the graphical programming language of SFCs in Section 2.2. An exemplary tank system is given in Section 2.3 which is modeled by SFCs. By using SFCs, it is possible to model the discrete controller behavior of a plant. Conditional ordinary differential equation (ODE) systems, which model the continuous behavior of a plant, are introduced in Section 2.4. How these conditional ODEs can be added to an SFC and the resulting HSFC are explained in Section 2.5. A formal definition of hybrid automata (HA) is given in Section 2.6. Furthermore, a transformation of HSFCs into HA is given. Afterwards, we are able to verify HSFCs by applying hybrid automata reachability analysis as presented in Section 2.7 to the transformed HSFCs. Additionally, we present the tool platform SpaceEx to analyze hybrid automata in Section 2.8.

2.1 Programmable Logic Controllers

PLCs are digital computers, which are used to control and regulate machines or plants. PLCs have multiple input and output interfaces and are built to be resistant to outside influences. E.g., a tank system, which consists of tanks, water pumps and valves, may be controlled by a PLC.

PLCs run a program by using scan-cycles (Figure 2.1). The scan-cycle can be split into four different phases. The first phase is the input scan phase, where the analog and digital input resulting from, e.g., sensors or user interaction are read and stored in the PLC's input memory table. The actual program is executed in the logic execution phase. The resulting values are written in the memory output table. In the output scan phase, the output memory tables values are written to the output module which are cards that typically have between 8 and 16 output interfaces. These interfaces are used to connect, for example, signal lights or displays. During the overhead phase diagnostics are

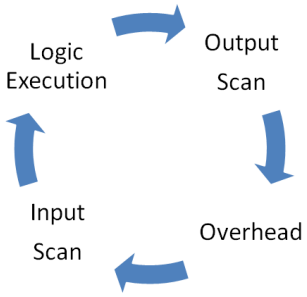

Figure 2.1: *The four phases of the PLC scan cycle*

performed, e.g., the watchdog timer is checked, which resets the PLC in case of a deadlock, and the memory is tested. This cycle is executed repeatedly until the PLC is turned off. Multiple programs running on a PLC are synchronized according to these cycles.

There exists an international standardization of five different PLC languages (IEC 61131-3 [IEC03]), which includes Function Block Diagrams (FBD), Ladder Diagrams (LD), Instruction Lists (IL), Structured Text (ST) and Sequential Function Charts (SFC). These languages can be used to program PLCs. IL and ST are similar to assembly languages, while FBD, LD and SFC are graphical programming languages. We only consider SFCs to model the programs of the PLC in the following sections as they allow us to present the control flow in a graphically and convenient, structured manner.

2.2 Sequential Function Charts

SFCs are used as a graphical programming language for programmable logic controllers. In this section we introduce this language and show how to use these charts to model an exemplary tank system in Section 2.3. We present the syntax of SFCs in Section 2.2.1 and the semantics in Section 2.2.2.

2.2.1 SFC Syntax

An SFC consists of steps which are connected by transitions. Actions blocks denote a tuple of actions and action qualifiers. They are attached to steps and the actions are executed according to their qualifiers.

The variables of an SFC are divided into three sets. $Var = Var_I \uplus Var_O \uplus Var_L$ is the set of variables, where Var_I are the input variables, Var_O the output variables and Var_L all local variables.

A step in an SFC can either be active or inactive. The initial step is active at the start of the execution. For other steps to become active, some conditions have to be met. A transition must be taken leading to the step, all sources of the transition have to be active, the transition guard must be satisfied and there must not be any enabled transitions with higher priority. If these conditions are satisfied, the next step becomes active and the source steps of the transition become inactive. Multiple steps can be activated this way at the same time since a transition can have multiple target steps, which are executed in parallel. Transitions in SFCs are urgent, which means that as soon as a condition of a transition is satisfied and all sources are active, a transition has to be taken. If multiple transitions can be taken at the same time, the transition with the highest priority is chosen. This priority is given by the partial order $\prec$.

Action blocks are tuples (q, a) consisting of a qualifier q and an action a. Action blocks can be assigned to each step of the SFC and the contained actions are executed during different events. An action can either be a variable assignment or an SFC. If an action a is an SFC, it becomes active, when the action is performed. If the SFCs history flag is active the nested SFC resumes at its last active step. In case the SFC uses no history, the nested SFC begins at its initial step. If the action a is a variable assignment, the value of the variable is changed accordingly. The qualifier specifies when the action will be executed. In the following examples, only the qualifiers entry, do and exit are used. These qualifiers are used instead of P1, N and P0. Some examples of SFC action qualifiers are listed in Figure 2.2.

	Qualifier Name	**Execution**
N/do	*Non-stored*	Action is executed while the step is active
S	*Set (stored)*	Action is executed until a reset occurs
R	*Reset*	Stops a *Set* action when the step becomes active
P1/entry	*Pulse, rising edge*	Executed once when the step becomes active
P0/exit	*Pulse, falling edge*	Executed once when the step is deactivated

Figure 2.2: *SFC action qualifiers*

Formally SFCs can be defined as described in Definition 2.1.

Definition 2.1 (Sequential Function Charts)
An SFC $C = (Var, Steps, Act, s_0, Trans, Blocks, \sqsubset, \prec, Hist)$ *is a tuple where*

- Var *is a finite set of variables*
- $Steps$ *is a finite set of steps*
- Act *is a finite set of actions referring to variables from Var in assignments and to SFCs whose variable and action sets are subsets of Var and Act resp. and whose action order is a subset of* $\sqsubset$
- $s_0 \in Steps$ *is the initial Step*
- $Trans \subset (2^{Steps} \setminus \{\varnothing\}) \times G^{Var} \times (2^{Steps} \setminus \{\varnothing\})$ *is a finite set of transitions, where* G^{Var} *is the set of all guards over the variables* Var *(transitions with multiple target/source steps define the begin/end of parallel branching)*
- $Blocks : Steps \to 2^{B_{Act}}$ *is a function which assigns a set of action blocks to each step, where* $2^{B_{Act}}$ *is the set of all action block sets*
- $\sqsubset \subset Act \times Act$ *is a total order on the actions*
- $\prec \subset Trans \times Trans$ *is a partial order on the transitions*
- $Hist \in \{0, 1\}$ *is a history flag*
  ($Hist = 1$*: SFC with history,* $Hist = 0$*: SFC without history)*

For a set T of transitions, let $target(T)$ be the target steps and $source(T)$ the source steps of T. We denote by $\bar{C}$ the set containing the SFC C and all its nested SFCs at all depths, $Steps(\bar{C})$ is the union of all steps and $Trans(\bar{C})$ the union of all transitions of these SFCs. SFC steps and transitions can be visualized as illustrated in Figure 2.3.

Steps consist of a name (StepA, StepB) as well as a set of action blocks. These action blocks are separated into three groups having the action qualifiers **entry**, **do** and **exit**. The actions $action_{a1}$, $action_{a2}$, ...and $action_{b3}$ are associated with the qualifiers above them. The condition $cond_t$ is assigned to the transition connecting the steps. Outgoing transitions always start at the bottom of an SFC step and incoming transitions are connected to the top of their target step. The initial step of an SFC is double-lined as seen in StepA.

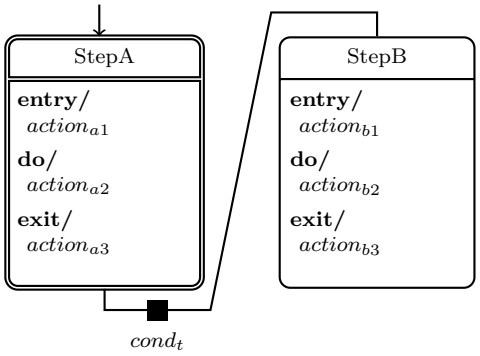

Figure 2.3: *Two step and a transition of an SFC*

2.2.2 SFC Semantics

The semantics of SFCs is described in this section. During a PLC cycle, the input data from the environment is updated and the variables are changed accordingly (input scan). In the logic execution phase, the set of transitions to be taken is determined and the transitions are executed. Afterwards, actions are determined and executed in order corresponding to the total action order $\sqsubset$. At the end of the cycle the output data is sent to the environment in the output scan phase. We omit the overhead phase in all following computations as the plant control is not affected by the diagnostics. An SFC configuration $(\sigma, readyS, activeS, activeA) \in \Sigma \times Steps(\bar{C}) \times Steps(\bar{C}) \times Act^*$ contains a state $\sigma : Var \to D$, which assigns a value to each, where D is the union of all data type domains.

$readyS$ is a set of ready steps and is the union of all active steps of the top-level SFC and the nested SFCs. Additionally, $readyS$ contains the last active steps of the currently inactive nested SFCs.

The set $activeS$ contains the active steps of the top-level SFC and of all nested SFCs, to which an active action points.

The sequence $activeA$ is a list of actions sorted by their priority. It contains the do actions of currently active steps, the exit actions of the source steps and the entry actions of the target steps of taken transitions. These actions are executed in the next PLC cycle.

$Conf$ is the set of all configurations. The initial configuration of an SFC is $(\sigma_0, \{s_0\}, \varnothing, \varnothing)$. The state σ_0 assigns an initial value to each variable. The configuration contains the initial step s_0 of an SFC C. The sets of enabled and taken transitions can be defined as shown in Equations (2.1) and (2.2) for a configuration $c = (\sigma, readyS, activeS, activeA)$.

$$enabled(c, C) = \{(S, g, S') \in Trans(\bar{C}) |\ S \subseteq activeS \wedge c \vDash g\} \tag{2.1}$$

$$taken(c, C) = \{t = (S, g, S') \in enabled(c, C) |\ \forall t_1 = (S_1, g_1, S_1') \in enabled(c, C).S \cap S_1 = \varnothing \vee t_1 \prec t\} \tag{2.2}$$

Assuming that σ is the updated input data from the environment and σ' are the values written to the output data at the end of the PLC cycle, the changes as shown in Definition 2.2 of the configurations can be observed, if the taken transitions and executed actions are considered.

Definition 2.2 (SFC Semantics)
The semantics of an SFC C is defined by the transition relation $\rightarrow \subseteq Conf \times Conf$ with $c = (\sigma, readyS, activeS, activeA) \rightarrow (\sigma', readyS', activeS', activeA') = c'$ if and only if

- $readyS' = (readyS \backslash source(taken(c, C))) \cup target(taken(c, C))$
- $(activeS', unsortedActiveA') = computeActiveSteps(readyS', \varnothing, \varnothing, C, c, activeA \cap \bar{C})$
- $activeA' = sort(unsortedActiveA', \sqsubset)$

The new set of ready steps can be computed by removing the source and adding the target steps of taken transitions. The new active steps are computed recursively. Starting at the top-level SFC, the algorithm recursively adds the active steps of the nested SFCs. The actions are sorted according to the total order given in the SFC by applying the function *sort*. The function *computeActiveSteps* which computes the set of active steps is given in [NA12].

2.3 Tank System

In this section, a simple tank system is presented as an example plant system. The plant controls are modeled using SFCs as introduced in Section 2.2. In addition to the components of the tank system, we introduce the dynamic behavior of the plant.

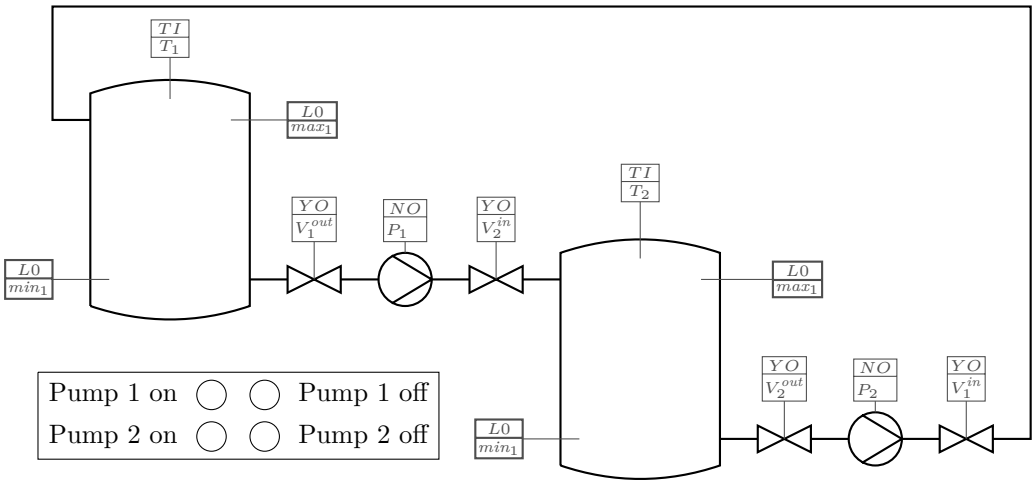

Figure 2.4: *The tank system and its control panel*

The tank system (Figure 2.4) regulates the water level of two different water tanks using pumps and valves. The water level in either tank may not exceed a maximum or fall below a minimum water level. Furthermore, each tank has two sensors which can detect whether the water level is still in its allowed range. Water can be pumped from each tank to lower its water level and increase the water level of the other tank in the process. Valves before and after each pump can be used to block the water flow. Additionally, the tank system includes an input panel to allow a user to manually turn the pumps on or off.

The tank system consists of two water tanks T_1 and T_2, valves V_1^{in}, V_1^{out}, V_2^{in} and V_2^{out} to allow or block water flow. Furthermore, the tank system has two water pumps P_1 and P_2, which are used to transfer water from T_1 to T_2 and from T_2 to T_1 at the rates c_1 and c_2. The water levels h_1 of T_1 and h_2 of T_2 change depending on whether a pump P_i is on (P_i) or off ($\neg P_i$) and the valves $V_i^{in/out}$ are open ($V_i^{in/out}$) or closed ($\neg V_i^{in/out}$). Assuming that both tanks are cylindrical and are built equally, and that if a pump is turned on then the corresponding valves are open, the following water level changes can be observed (Equations (2.3) to (2.6)).

$$\neg P_1 \wedge \neg P_2 : \dot{h}_1 = 0, \dot{h}_2 = 0 \tag{2.3}$$

$$P_1 \wedge \neg P_2 : \dot{h}_1 = -c_1, \dot{h}_2 = c_1 \tag{2.4}$$

$$\neg P_1 \wedge P_2 : \dot{h}_1 = c_2, \dot{h}_2 = -c_2 \tag{2.5}$$

$$P_1 \wedge P_2 : \dot{h}_1 = c_2 - c_1, \dot{h}_2 = c_1 - c_2 \tag{2.6}$$

The sensors min_1 and max_1 check whether the water has fallen below or exceeded a specific level for T_1. In T_2 the sensors min_2 and max_2 work analogously. The switches P_i^{on} and P_i^{off} are used to manually turn the pumps P_i on and off. In

order to prevent the tanks from running dry or overflowing, the pumps may only be turned on if the sensors detect an appropriate water level. In case of an overflowing tank, the water level remains at the tanks maximum water level as the surplus water cannot be held by the tank. A pump is turned off automatically, if a the corresponding tank is running dry.

The history flag is not considered in the following examples, since we do not use nested SFCs. If the dynamic behavior of the tank system is neglected, two SFCs describe the control flow of the example tank system (see Figure 2.5).

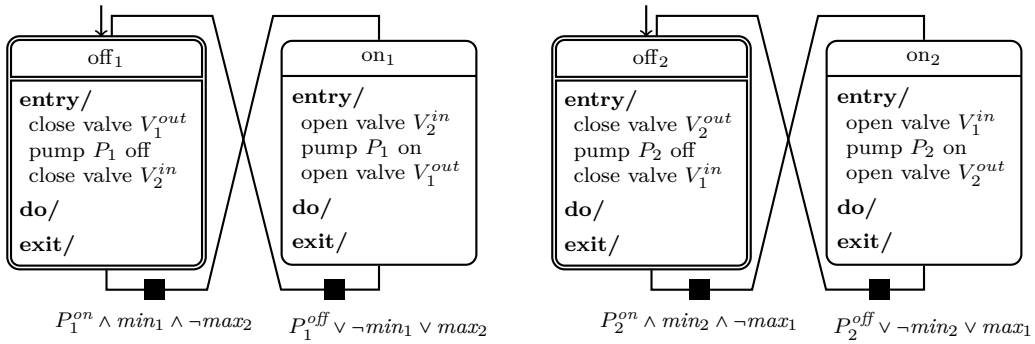

Figure 2.5: *SFC of the tank system*

The two given parallel SFCs are components that model the tank system. Furthermore, two steps are needed for each pump, since in both systems, the pump is either turned on or off. Once a step is entered the associated valves are opened or closed and the pump is turned on or off. Both SFCs are executed in parallel allowing the state changes of the two pumps simultaneously. A step change occurs when a pump is turned on if it was previously off, the water level is high enough and the second tank is not full or when a previously running pump is turned off, drains the tank or the other tank is overflowing.

2.4 Conditional ODE Systems

In order to model the dynamic behavior of a plant, we use conditional ODE systems. A conditional ODE system $condODE_{sys}$ consists of a condition ODE_{cond} and associated differential equations ODE_{list}. Each ODE system $condODE_{sys}$ models the behavior of the plant dynamics under the given condition.

An equation of the form $\dot{v} = c$ describes the behavior of the continuous variable v over time. In general, we also allow linear ODEs. c is the derivate of the function, which represents the dynamic behavior of v. In the following examples, we omit the valves and assume they are opened when the corresponding pump is

running and closed if it is turned off. The dynamic behavior of the example tank system can be modeled as the conditional ODE systems shown in Equations (2.7) to (2.10).

$$(\ P_1 \wedge \neg max_2 \wedge \ min_1) \wedge (\neg P_2 \vee \ max_1 \vee \neg min_2) : \dot{h}_1 = -c_1 \quad \dot{h}_2 = c_1 \tag{2.7}$$

$$(\neg P_1 \vee \ max_2 \vee \neg min_1) \wedge (\ P_2 \wedge \neg max_1 \wedge \ min_2) : \dot{h}_1 = c_2 \quad \dot{h}_2 = -c_2 \tag{2.8}$$

$$(\ P_1 \wedge \neg max_2 \wedge \ min_1) \wedge (\ P_2 \wedge \neg max_1 \wedge \ min_2) : \dot{h}_1 = c_2 - c_1 \quad \dot{h}_2 = c_1 - c_2 \tag{2.9}$$

$$(\neg P_1 \vee \ max_2 \vee \neg min_1) \wedge (\neg P_2 \vee \ max_1 \vee \neg min_2) : \dot{h}_1 = 0 \quad \dot{h}_2 = 0 \tag{2.10}$$

In this model both pumps may only be activated when the tanks are not full to protect them from overflowing. This condition can be improved if we know whether $c_1 \leq c_2$ or $c_1 \geq c_2$, since this allows us to see which tank will be filled and which will be drained, when both pumps are running.

In the following computations a reduced model of the dynamic behavior is used. The sensor checks of both tanks are omitted. Only the state of the pumps is examined in the condition and influences the continuous behavior of h_1 and h_2. Additionally, we use the values $c_1 = 1$ and $c_2 = 2$ as the water level changes, if water is pumped from one tank to another. The disjoint conditional ODE systems in Equations (2.11) to (2.14) model the behavior of the water levels.

$$\neg P_1 \wedge \neg P_2 : \dot{h}_1 = \ 0, \dot{h}_2 = \ 0 \tag{2.11}$$

$$P_1 \wedge \neg P_2 : \dot{h}_1 = -1, \dot{h}_2 = \ 1 \tag{2.12}$$

$$\neg P_1 \wedge \ P_2 : \dot{h}_1 = \ 2, \dot{h}_2 = -2 \tag{2.13}$$

$$P_1 \wedge \ P_2 : \dot{h}_1 = \ 1, \dot{h}_2 = -1 \tag{2.14}$$

Non-disjoint conditional ODE systems are also possible, but using them requires prioritizing the conditional ODE systems. Non-disjoint conditions are not considered in the following examples. In either case, the ODE system with the first condition, which is satisfied, determines the current dynamic behavior. If no condition holds, we assume chaotic behavior for the continuous variables.

By combining the conditional ODE systems with an SFC we can construct an HSFC. In this HSFC, each step has conditional ODE systems attached to it, which define the continuous growth of the variables under the given conditions.

2.5 Hybrid Sequential Function Charts

HSFCs [NA12] are SFCs, which additionally consider the behavior of continuous variables. Plant dynamics are modeled by conditional ODE systems which are attached to the steps of the SFC. Multiple conditional ODE systems can be assigned to each step. In case the condition of an ODE system is satisfied the differential equations describe how the continuous variables change over time.

The set of variables Var is now extended by continuous variables Var_C. The set of variables of an HSFC is defined as $Var = Var_I \uplus Var_o \uplus Var_L \uplus Var_C$. A new function Dyn is introduced to assign conditional ODE systems to the steps of the HSFC. Let $CODE_{Var_C}$ be the set of all conditional ODE systems over the continuous variables Var_C. The function in Equation (2.15) assigns a sequence of conditional ODE systems to a step.

$$Dyn : Steps \to CODE^*_{Var_C} \tag{2.15}$$

Taking the conditional ODE system and the SFC model of the tank system, which have been defined in the previous section, an HSFC can be created. The complete HSFC, where each step has been assigned all conditional ODE systems, for the $tank_1$ can be constructed as seen in Figure 2.6.

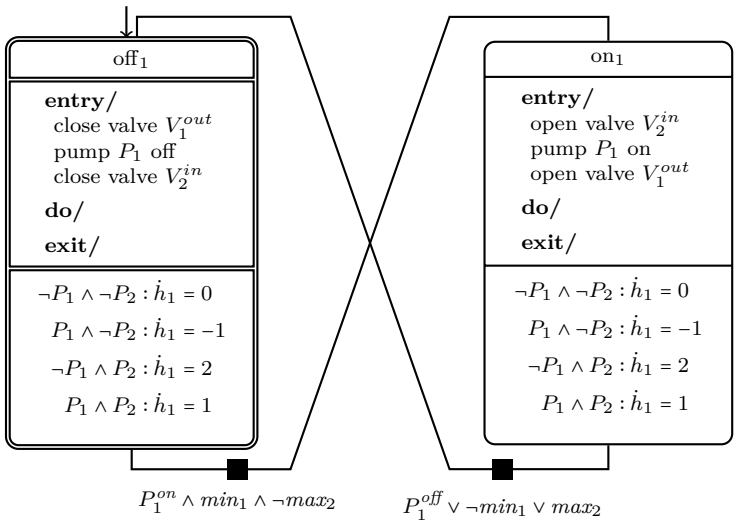

Figure 2.6: *SFC of $tank_1$ is extended into an HSFC by conditional ODE systems*

The HSFC of $tank_2$ can be constructed analogously. Adding all conditional ODE systems to each step is not reasonable, since some conditions are not satisfiable for some steps. For example $P_1 \wedge \neg P_2$ is never satisfied in step off_1 of

$tank_1$ as the pump P_1 is always off in this step. A reduced system denotes a system which does not have all conditional ODE systems attached. Verifying reduced systems, which do not model the PLC completely, is faster and may already prove the complete model to be correct. In the following sections those HSFCs which do not have all available conditional ODE systems attached to each step, are considered as well.

An HSFC configuration $(\sigma, readyS, activeS, activeA, activeD)$ is similar to the SFC configuration, but is extended by the set $activeD$. The set is the set of active ODEs, which are the ODEs, that are attached to active steps. An ODE is only in $activeD$, if its condition is the first satisfied condition in a step in the sequence of all attached conditional ODE systems.

During a PLC cycle, the variables evolve according to the active ODEs. This dynamic evolution extends the SFC semantics in order to create HSFC semantics. Assuming a time elapse of t with $\delta_l \leq t \leq \delta_u$, where δ_l and δ_u are the minimal and maximal cycle time. Additionally the values of all continuous variables at $t = 0$ must correspond to the values in $\sigma\,|_{Var_C}$. For conditional ODE systems $condODE_{sys} = (ODE_{cond}, ODE_{list})$, the HSFC semantics are defined as shown in Definition 2.3.

Definition 2.3 (HSFC Semantics)
Runs of HSFCs consist of the alternating execution of steps of the embedded SFCs and time steps $(\sigma, readyS, activeS, activeA, activeD) \rightarrow (\sigma', readyS, activeS, activeA, activeD')$ *with*

- $activeD' = \bigcup_{s \in activeS}\{d \mid \exists i \in \{1, \ldots, n\}:$
  $Dyn(s) = ((\mathrm{ODE}_{c1}, \mathrm{ODE}_{l1}), \ldots, (\mathrm{ODE}_{cn}, \mathrm{ODE}_{ln})) \wedge d \in \mathrm{ODE}_{li}$
  $\wedge\, \sigma \vDash \mathrm{ODE}_{ci} \bigwedge_{j=1}^{i-1} \sigma \nvDash \mathrm{ODE}_{cj}\}$,
- $\sigma'(v) = \sigma(v)$ *for all* $v \notin Var_C$ *and* $\sigma'\,|_{Var_C} = f(t)$ *for some* $\delta_l \leq t \leq \delta_u$ *and* f *a solution to ODE systems in activeD with* $f(0) = \sigma\,|_{Var_C}$.

The definition of HSFC and its transformation are adopted from the paper "Hybrid Sequential Function Charts" [NA12], which describes HSFCs in more detail.

2.6 Hybrid Automata

Hybrid automata [ACH+95] are automata that can model discrete as well as continuous behavior of systems. HA are formalized in Definition 2.4.

Definition 2.4 (Hybrid Automaton)
A hybrid automaton $\mathcal{H}$ is a tuple (Loc,Var,Edge,Act,Inv,Init)

- *Loc is a finite set of locations*
- *Var is a finite set of real-based variables; A valuation $\nu \in V, \nu : Var \to \mathbb{R}$ assigns a value to each variable; A state $s \in Loc \times V$ is a location valuation pair*
- *$Edge \subseteq Loc \times 2^{V^2} \times Loc$ is a set of edges*
- *Act is a function assigning a set of time-invariant activities $f : \mathbb{R}_{\geq 0} \to V$ to each location, i.e., $\forall l \in Loc : f \in Act(l) \implies (f + t) \in Act(l)$ 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*

Each edge consists of a source location and a target location as well as a guard and an effect. An edge can only be taken if the guard is satisfied and the invariant of the target location is satisfied after applying the effect of the edge to the current variable valuation. Immediately after an edge is taken the values of the variables in Var are updated according to the effect. The activities in each location describe how the continuous variables change over time. An invariant is a guard, which needs to be satisfied to allow the hybrid automaton to stay in a location or to enter a location, which is connected by an edge to the current location.

There are two different kinds of steps, namely discrete steps (jumps) and time steps (flows), in hybrid automata. Discrete steps allow the automaton to change from one location to the next if a corresponding edge exists and can be taken. A discrete step may not enter a location if the invariant of that location is not satisfied. Assuming $l, l' \in Loc$ and $v, v' \in V$, a discrete step is defined in Equation (2.16).

$$\frac{(l, (v, v'), l') \in Edge \quad v' \in Inv(l')}{(l, v) \to (l', v')} \tag{2.16}$$

Time elapses are modeled by time steps. A time step causes the continuous variables to change according to the function of the current locations activities. The activities for each location are usually given as ODE systems. A time step

may not violate the locations invariant by evolving. Assuming $l \in Loc$, $v, v' \in V$ and time t, a time step is defined in Equation (2.17).

$$\frac{f \in Act(l) \quad f(0) = v \quad f(t) = v' \quad t \geq 0 \quad f([0,t]) \subseteq Inv(l)}{(l,v) \xrightarrow{t} (l,v')} \tag{2.17}$$

If neither a discrete or timed step can be taken in the current state of the hybrid automaton, the system is in a deadlock.

2.6.1 Transformation of SFC to HA

The transformation of an HSFC into an HA is based on the transformation of an SFC. Consequently, we begin our explanation of the HSFC transformation by converting an SFC into an HA. To transform an SFC into a hybrid automaton, for each step of the SFC a location in the hybrid automaton is created. If the step of the SFC is initial the corresponding location in the hybrid automaton defines the initial states. Figure 2.7 illustrates how SFC steps and transitions are converted into a hybrid automaton.

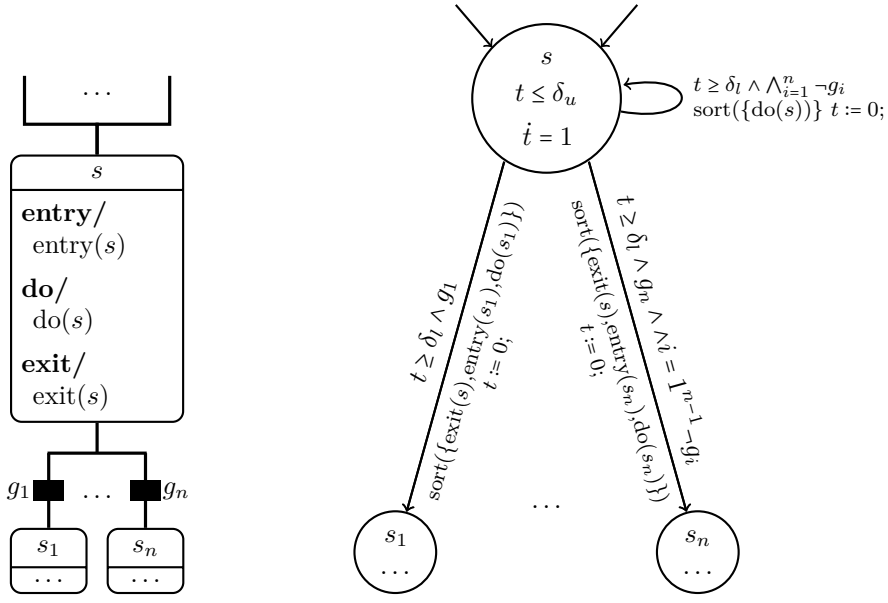

Figure 2.7: *Transformation of a SFC into a hybrid automaton*

In order to model the time elapse between two PLC scan cycles, a continuous variable is introduced to the hybrid automaton. This continuous variable t is a clock variable, which keeps the cycle time from exceeding an upper bound δ_u by adding the invariant $t \leq \delta_u$ to each location. Since in every PLC cycle at least some time has to pass and to avoid Zeno behavior [ACH+95] in the

hybrid automaton, where infinitely many discrete steps are taken while only a finite amount of timed steps are used, a guard is added to each transition. This guard ensures that a time of at least δ_l has passed before the next cycle. The duration of each PLC cycle is between $\delta_l \leq t \leq \delta_u$. The time t is reset after each transition.

The transitions of the SFC are also translated to the hybrid automaton. The active actions are sorted by the given action order $\prec$. These actions are derived from the source and target step of the transition. The exit actions are taken from the source step, while the entry and do actions are derived from the target step. The actions of the SFC are moved from the steps to the transitions of the HA during the transformation since we can only model these actions as effects of HA transitions in the resulting automaton. Besides all transitions of the SFC, each location has an additional self loop which only contains the sorted do qualified actions of the step. This transition simulates the PLC staying in the same step executing the do actions during a PLC cycle.

The transition priority $\sqsubset$ is maintained by adding the negations of all guards of transitions with a higher priority than the current transition. The additional self loop may only be taken, if all other guards are not satisfied. This is due to the urgency of transitions in an SFC.

2.6.2 Transformation of HSFC to HA

SFCs extended by conditional ODE systems are HSFCs. Consequently, we need to extend the transformation of SFC as presented in Section 2.6.1 to incorperate these conditional ODE systems. We do not consider nested SFCs or parallel transitions, but the transformation can be adapted to the general case.

In addition to the SFC transformation, each conditional ODE system of the HSFC results in new locations. Each step in the HSFC with n conditional ODE system attached is transformed into $n+1$ locations in the hybrid automaton. These $n+1$ ODE locations are pairwise connected by transitions without guards and all share the same transitions as the original step. The condition of the ODE system in each location is conjuncted with its invariant and the ODE are added to the activities of the location.

In the additional $(n+1)$th location (default location), all continuous variables exhibit chaotic behavior and the negated conditions of all ODE systems are conjuncted with the locations invariant. Thus, this step is only reached if no conditional ODE system is satisfied meaning no statement about the continuous variables can be made. In the following we do not consider this $(n+1)$th location as our current verification process is not able to create this location.

The transitions which allow switching between locations do not have any guards to allow the system to instantly switch locations if the condition of the current ODE system has become false. In case the initial step of the HSFC has multiple conditional ODE systems attached to it, all resulting hybrid automaton locations are part of the initial states. Figure 2.8 shows how aN HSFC step is transformed into several HA locations.

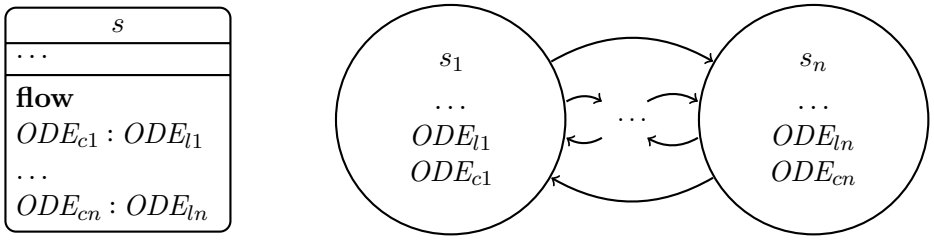

Figure 2.8: *Transformation of a HSFC into a hybrid automaton*

Since we assume disjoint conditional ODE systems, we do not need to add the negation of the other conditional ODE systems of the step in order to prioritize them. A priority order could be achieved by adding ODE_{c1} to the first, $\neg ODE_{c1} \wedge ODE_{c2}$ to the second, ... and $\neg ODE_{c1} \wedge \ldots \wedge \neg ODE_{c(n-1)} \wedge ODE_{cn}$ to the last location. In this case the priority beginning with the highest would be ODE_{l1}, ODE_{l2}, ..., ODE_{ln}. Considering an HSFC with m steps and n conditional ODE systems, which can be attached to each step, the total number of locations is the product $n * m$. Both transformations are described in more detail in [NA12].

2.7 Reachability Analysis

In order to verify hybrid automata, we need to compute their reachability. The problem of reachability for hybrid automata in general is undecidable. Considering linear hybrid automata, we are able to efficiently compute bounded reachability meaning reachability within a fixed number of steps [ACH+95].

The forward analysis computes the reachability starting from the initial states. Consequently, safety conditions can be verified by examining the reachable states. The condition is given in the form of forbidden states, i.e, states which should not be reached during the analysis. If any of these states is reachable, the hybrid automaton is not safe under the given safety conditions. The backwards analysis starts by computing the backwards reachability of the forbidden states, i.e., the states which do not satisfy the given set of safety conditions. If any initial state is backwards reachable from the given safety conditions, the hybrid automaton is not safe under the given conditions.

Both approaches compute the reachability of hybrid automata. The forward (backward) analysis terminates if either forbidden (initial) states are reachable or a fixed-point meaning a point, where no new states, i.e., states which have not been reached during the previous analysis, is found. In order to successfully analyze hybrid automata approximation and minimization can be used [ACH+95]. We employ the third-party tool platform SpaceEx to perform forward reachability analyses for hybrid automata .

2.8 SpaceEx - State Space Explorer

The safety verification of a hybrid automaton is accomplished by applying SpaceEx [FLGD+11], a tool used for reachability and safety analysis. This reachability analysis is an over-approximation of the reachability. Our transformation creates an XML-File containing the HSFC after its transformation into a hybrid automaton and an additional CFG-File containing the configuration applied during the SpaceEx execution. In this configuration file, the initial variable ranges and locations, the forbidden states, the output format, the number of iterations and other preferences are defined. An advantage of SpaceEx is, that there are two different verification scenarios available. The PHAVer scenario uses the PHAVer tool [Fre05] and is applicable on linear hybrid automata. Furthermore, this scenario computes the exact results for piecewise constant flows. The LGG Support Function scenario, which implements a variant of the Le Guernic-Girard (LGG) algorithm [GG09], is also available.

Since reachability analysis for hybrid automata is undecidable in general, SpaceEx can set a maximum number of iterations. These iterations represent the number of discrete post computations on the cross product of locations and variable valuations. In case a negative number is set as the number of iterations, the reachability analysis only terminates, if a fixed point is found. Additionally, ranges for time steps can be set. If these configurations are chosen too cautiously, SpaceEx may produce a meaningless result, due to the computation of too few reachable states.

Different SFCs can run in parallel on a PLC. Consequently, we obtain different hybrid automata components as well. There are two approaches to connect these components in order to analyze the given model. The first approach the composition of these components is computed before the analysis starts. SpaceEx, however, uses an on-the-fly approach, thus, the compositions are computed at the moment they are required. Consequently, only the reachable parts of the automaton are created in memory.

SpaceEx is able to return several types of outputs. The INTV output creates a file containing intervals, which represent the range of values for each variable in the model. In addition, SpaceEx generates a TXT-File which contains state information and the vertices of the polytopes of the reachable area of the continuous variables. The state information consist of the initial values and locations.

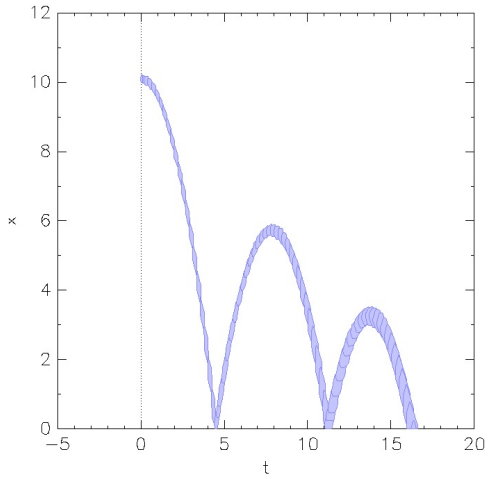

Figure 2.9: *2d SpaceEx output*

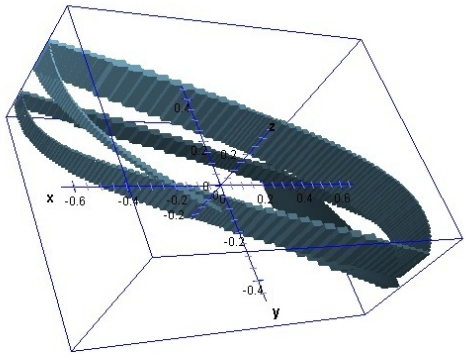

Figure 2.10: *3d SpaceEx output*

Possible graphical outputs are the GEN (Vertice List) and the JVX output [JP00]. Both outputs are presentable by third-party tools like graph of the Plotutil package [MT00] for GEN and JavaView [Pol06] for the JVX format. Both formats support up to three dimensions as seen in Figures 2.9 and 2.10.

The TXT-File and the graphical outputs contains the complete verification process if a safety verification was successful, but only contains the reachable forbidden states if the model is unsafe. The INTV-File contains the ranges of the complete reachability analysis.

The CEGAR-based plant control verification uses TXT and INTV outputs to analyze a counterexample of a model after a failed safety verification. These output informations are used to find a reason for the unsafetyness of the current model and to choose an appropriate refinement.

2.9 Summary

In this chapter, programmable logic controllers are introduced as controllers for plants. To verify the controller program, SFCs are used to model the plants discrete behavior. An example of such a model is given for the example tank system. The dynamic behavior of a plant can be added to the SFC in form of conditional ODE systems. Verifying the resulting HSFC allows us to verify the plant control, meaning its discrete and continuous behavior. Conditional ODE systems are provided for the example tank system and combined with the SFC of the system in order to create an HSFC. Since we want to use hybrid automata verification to show the correctness of a plant, hybrid automata are defined. Transforming SFC to HA additionally allows us to extend the initial SFC model. The transformation of SFCs into HA is explained. Existing tools to verify hybrid automata may be applied to prove the correctness of the SFC. Additional transformations are included during the transformation of HSFCs to HA. Hybrid automata can be verified using existing verification tools, thus verifying the plant control. This is accomplished by reachability analysis of hybrid automata. Therefore, an introduction to this reachability analysis is given and the tool platform SpaceEx which we use to perform this analysis is presented.

Chapter 3

CEGAR-based plant control verification

In this chapter, the process of verifying hybrid sequential function charts using a verification tool for hybrid automata is explained. Figure 3.1 shows the steps of the transformation and of the verification which is described in the following sections.

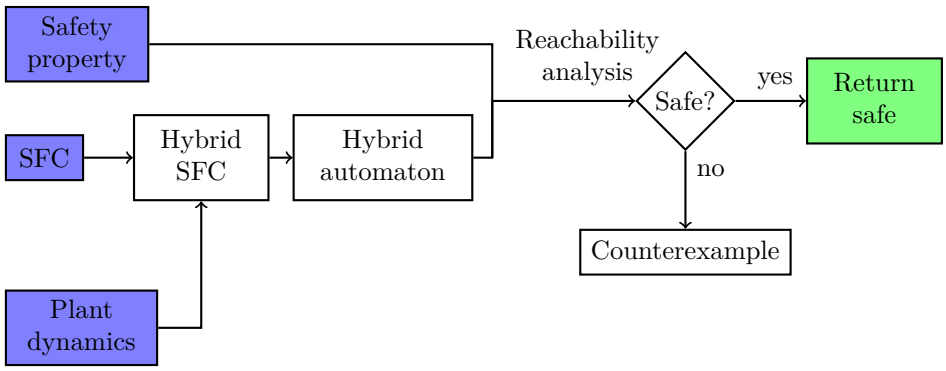

Figure 3.1: *Transformation and verification of a SFC and plant dynamics*

Figure 3.1 describes the following parts of the our verification. The input of the plant verification consists of the plant control in the form of SFCs, the plant dynamics and a set of safety properties. The plant control and dynamics are combined into an HSFC. This HSFC is transformed into an HA which is verified using SpaceEx and the given safety condition. During this transformation additional changes to the hybrid automaton are made to create a viable SpaceEx model. In case the SpaceEx verification succeeds, the model is safe. We analyze the counterexample provided by SpaceEx, if the verification fails.

The CEGAR-based plant verification iteratively adds conditional ODE systems to an HSFC. During each iteration, the HSFC is verified for a given safety condition. In case this verification fails, a refinement strategy chooses one or multiple conditional ODE systems which are added to the HSFC. Afterwards, the next iteration begins by verifying the extended HSFC. This refinement is described in Chapter 4.

Firstly, the steps of the the CEGAR-based plant control verification are described in Section 3.1. Hereinafter, we introduce the three parts of the verification input in Section 3.2. The Beremiz PLC Open Editor [Tis12] to create SFCs is introduced, which is used to model systems like the example tank system (see Figure 2.4) in Section 2.3. In addition to modeling the tank system, a custom syntax for conditions of the SFC transitions and ODE conditions is presented. Furthermore, an XML format is given to store conditional ODE systems. This format allows the assignment of conditional ODEs to steps of an HSFC. Additionally, safety conditions in the form of forbidden states are introduced. The combination of the SFC model and the conditional ODE systems represents the HSFC of the plant.

The tool platform SpaceEx [FLGD+11] to verify hybrid automata is shown in Section 3.3. Reachability analysis can be used to check whether a given automaton satisfies a safety condition. Methods to circumvent the restrictions of SpaceEx are implemented in order to correctly verify the transformed automata without changing the semantics of the model. During the verification, the transformation of an HSFC is performed. This transformation is extended to include additional components simulating, e.g., user inputs and total elapsed time. The result of the safety verification determines whether the model satisfies the given safety property or not. In case the model is unsafe in our CEGAR-based verification approach, we further examine whether it is refinable by attaching additional conditional ODE systems to HSFC steps. Moreover, the output formats of SpaceEx are elaborated in Section 3.4. They describe the counterexamples of a failed verification.

3.1 CEGAR Plant Control Verification

In this section, we describe the procedure of our CEGAR-based plant control verification. The plant control verification constitutes an iterative approach. It transforms one or multiple SFC model of a plant and given conditional ODE systems into hybrid automata. This automaton is translated into the SX language [CFL10] and stored in an XML-File, which can be parsed by SpaceEx. In addition to the automaton, a configuration file is generated specifying the settings of SpaceEx. Finally, SpaceEx tries to verify the hybrid automaton using this given configuration.

The SpaceEx [FLGD+11] safety verification yields one of two possible results for a given model and configurations. Either the safety verification succeeds or fails depending on whether forbidden states are reachable. At the beginning of the verification none or only few conditional ODE systems are added to the HSFC. This reduced model is much smaller than the complete representation,

therefore SpaceEx is likely to verify the system much faster as well. In case SpaceEx is able to verify the reduced model, the complete model safe.

If forbidden states are reachable and the model can be further refined, the system is refined iteratively until the safety verification is either successful or fails and no more conditional ODE systems can be added to the current model.

Based on these refinement iterations, the safety verification may have to run several times. Every time the model has been refined, the next iteration of the verification starts by transforming the refined HSFC into a hybrid automaton.

3.2 Plant Control Verification Input

In this section the three parts of the input of the CEGAR-based plant control verification are introduced. We present the third-party tool Beremiz PLC Open Editor [Tis12] to create SFCs in Section 3.2.1. Additionally, we propose a format to independently store conditional ODE systems as described in Section 3.2.2. Furthermore, the format of safety conditions in SpaceEx is presented in Section 3.2.3.

3.2.1 Plant Control

SFCs for PLCs are created by using an appropriate editor, in our case, the Beremiz PLC Open Editor [Tis12]. The PLC Open standard focuses around the IEC 61131-3 [IEC03] standard, as it is the only global standard for industrial control programming. By using Beremiz, we conform to these standards while creating models for PLCs. The PLC Open standard contains a definition for the five programming languages. All five of these languages can be used to model PLCs in the editor. Furthermore, Beremiz allows the user to create multiple programs for a PLC. These components can be modeled by SFCs. We use Beremiz to specify the discrete plant control for our CEGAR-based verification. The tank system is presented in Figure 2.4 in Section 2.3.

As shown in Figures 3.2 and 3.3, each component is composed of two steps (on_1, off_1 and on_2, off_2), which represent the state of each pump. In these models only entry (pulse, rising edge) qualified actions are attached to the SFC steps. If step on_1 is entered, pump P_1 is turned on and the valves V_1^{out} and V_2^{in} are opened. These valves are closed and the pump is turned off if the off_1 step is entered. Analogously the valves V_2^{out}, V_1^{in} of P_2 are opened and closed when on_2 and off_2 of the second component are entered. The transition guards $Switch_{on}$ and

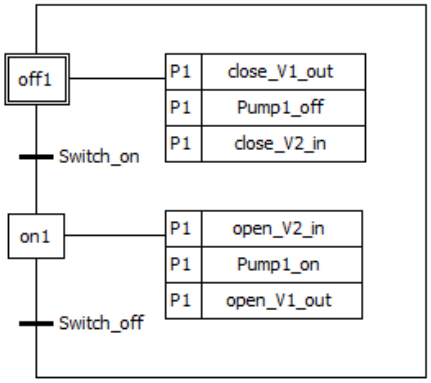

Figure 3.2: *Beremiz model of $tank_1$*

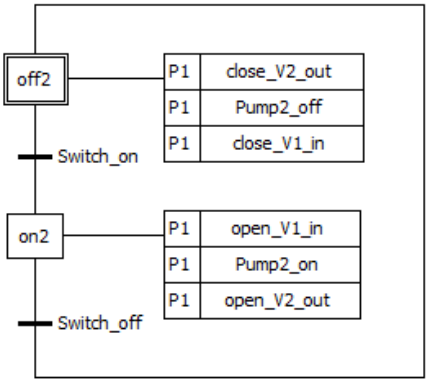

Figure 3.3: *Beremiz model of $tank_2$*

$Switch_{off}$ of the first component evaluate the user input P_1^{on} and P_1^{off} as well as the sensors min_1 and max_2. In the second component, the same transition guards are applied for P_2^{on}, P_2^{off}, min_2 and max_1 as illustrated in Figure 2.5 in Section 2.3.

The language of transition guards and actions in the editor is represented as Structured Text (ST). Though Structured Text is used, guards are written using a custom syntax for these conditions, which is explained in the following. The syntax supports basic logic operators and comparators to check integer and real values. The syntax is listed in Figure 3.4.

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

Figure 3.4: *Custom condition syntax*

The syntax is defined as `TransitionName := Condition` for SFC transition guards. The transition guard $Switch_{on}$ of $tank_1$ checks for $P_1 \wedge min_1 \wedge \neg max_2$ by applying the command `Switch_on := chkb_P1_on AND min1 AND NOT max2`. The actions to open and close a valve or to start and stop a pump are implemented by assigning values of 1 and 0 to the variables representing true and

false. For instance, the execution of command `V1out_open := 0;` closes the valve V_1^{out}. These variable assignments are commands for the actuators of the plant. We do not consider nested SFCs as described in Section 2.6.2. The SpaceEx analysis tool is not able to handle disjunctions and the strict operators > and <. In Section 3.3 we propose methods to circumvent these restrictions.

3.2.2 Plant Dynamics

The plant dynamics are stored in an XML-File, which contains all specified ODE systems and their conditions as well as the initial values of the continuous variables. Each ODE system is stored in a separate tag `<condODE>` with at least two children. The first required child is a tag `<cond>` containing the condition of the ODE system. The conditions are saved as logic formulas using the same syntax as the SFC guards (see Figure 3.4). An exemplary condition $A \wedge B$ is stored as `<cond>A AND B</cond>`. Alternatively, these conditions can be saved as CDATA sections in order to increase their readability in case inequality comparators are used. $x \geq 0$ corresponds to `<cond>x &gt;= 0</cond>` and `<cond><![CDATA[x >= 0]]></cond>` in the XML-File. To represent the equations of the ODE system, `<condODE>` contains one additional child tag `<equation>` for each equation of the system. The ordinary differential equation $\dot{h} = 1$ corresponds to `<equation>h'==1</equation>`.

The initial values of the continuous variables are stored in the file using a tag called `<init>`. These can be used to overwrite the initial values given in the SFC model. Thus, the user is able to change the initial variable assignments of the continuous variables without changing the SFC model. Each variable assignment requires an additional tag. These new tags `<value>` have a property `var` containing the name of the variable which is assigned. The tag itself contains the new initial value for this variable. The previously given reduced conditional ODE systems for the two tank example can be stored in an XML as demonstrated in Listing 3.1.

The variables `chkb_P1_on` and `chkb_P2_on` correspond to P_1 and P_2. Since these conditions are added as an invariant to the steps of the hybrid automaton and SpaceEx is not capable to parse disjunctions, no logic OR may be used within the conditions. Additional transformations to circumvent this restriction will be implemented as future work.

```
<?xml version="1.0" encoding="UTF-8"?>
<condODEsys>
  <condODE>
    <cond>NOT chkb_P1_on AND NOT chkb_P2_on</cond>
    <equation>h1' == 0</equation>
    <equation>h2' == 0</equation>
  </condODE>
  <condODE>
    <cond>NOT chkb_P1_on AND chkb_P2_on</cond>
    <equation>h1' == 2</equation>
    <equation>h2' == -2</equation>
  </condODE>
  <condODE>
    <cond>chkb_P1_on AND NOT chkb_P2_on</cond>
    <equation>h1' == -1</equation>
    <equation>h2' == 1</equation>
  </condODE>
  <condODE>
    <cond>chkb_P1_on AND chkb_P2_on</cond>
    <equation>h1' == 1</equation>
    <equation>h2' == -1</equation>
  </condODE>
  <init>
    <value var="h1">25</value>
    <value var="h2">25</value>
  </init>
</condODEsys>
```

Listing 3.1: *Conditional ODE systems*

The possibility to add conditional ODE systems to specific steps of an HSFC is realized by attributes added to each `<condODE>` tag accordingly. The attribute name denotes the name of a component and its value constitutes a list of steps the ODE system is added to. A step is refined by a conditional ODE system if the conditional ODE system has been attached to the step.

```
<condODE tank1="off1,on1" tank2="off2">
  <cond>NOT chkb_P1_on AND NOT chkb_P2_on</cond>
  <equation>h1' == 0</equation>
  <equation>h2' == 0</equation>
</condODE>
```

Listing 3.2: *Conditional ODE system refining steps*

In Listing 3.2 the conditional ODE system has already been attached to the step on_1, off_1 of component $tank_1$ and off_2 of component $tank_2$.

3.2.3 Safety Condition

Since our goal is the verification of plants, we need to specify conditions which are verified. The safety conditions the model has to satisfy can be set in the

configurations of SpaceEx. Forbidden states can be defined in this CFG-File by conjunction or disjunction. In addition to ==, ≤ and ≥, SpaceEx is able to parse >, < and parenthesis. The forbidden states can be defined as non-linear convex constraints in contrast to the transition guards and location invariants. In case these forbidden states are reached during the SpaceEx analysis, the model does not satisfy the safety condition. For example, the condition $(x > 5 \vee x < 0) \wedge y > 0$ corresponds to the forbidden states `(x > 5 | x < 0) & y > 0`. Appropriate forbidden states for the example tank system are shown in Equation (3.1).

$$(h_1 < 10) \vee (h_1 > 40) \vee (h_2 < 10) \vee (h_2 > 40) \tag{3.1}$$

The corresponding forbidden states in the SpaceEx configuration file are defined as `h1 < 10 | h1 > 40 | h2 < 10 | h2 > 40`. These forbidden states are equivalent to the safety assumption, that the water levels of the tanks will remain between 10 and 40. A forbidden location `locA` for a component `compA` can be defined as `loc(compA)==locA`. Forbidden locations are not considered in this thesis.

3.3 SpaceEx Analysis

In this section, we elaborate on the restrictions of SpaceEx and the transformation into a viable model, i.e., a model, which can be parsed by SpaceEx. SpaceEx version 0.9.5 [FLGD+11] has several compatibility restrictions regarding model specifications, which have to be circumvented in order to check the example tank system (see Figure 2.5) in Section 2.3.

For instance, boolean variables are not supported by SpaceEx. Thus, they have to be transformed into integer values, which contain either 0 (*false*) or 1 (*true*). Furthermore, the data range of the interval output is correctly computed for continuous variables only. During the transformations from HSFCs to HA, all read variables v_i where $i \in \{1, \ldots, n\}$, are added as continuous variables by assigning an ODE to each equation, which does not change the values of the variables over time ($\dot{v}_i == 0$).

As SpaceEx is not capable of parsing disjunctions, the guard of each transition is transformed into disjunctive normal form (DNF). During the transformation to an HA, we generate one transition for each conjunctive term, thus, eliminating the disjunction in the resulting model. Furthermore, parsing >, < and ¬ is not possible, while using SpaceEx. As SpaceEx already over-approximates the reachability, we replace > by ≥ and < by ≤. Consequently, this over-approximation affects the negation of conditions. For example, the conditions $\neg(v_1 > v_2)$ and $\neg(v_1 < v_2)$ are converted into $v_1 \leq v_2$ and $v_1 \geq v_2$. Additionally, $\neg(v_1 == v_2)$

is converted to *true*. A boolean variable b can be checked by b or $\neg b$. These evaluations are translated into $b == 1$ and $b == 0$ as boolean variables are either 1 or 0 as described previously.

Furthermore, shared input variables of different hybrid automata are updated instantly for each component. This immediate update changes the semantics of SFCs as the input variables are only updated at the start of a PLC cycle as described in Section 2.1. We introduce a new global variable corresponding to the local input variables in each component. The global variable is updated instantly, while its value is copied into the local input variable at the start of every PLC cycle. Additionally, we present new locations (*synch* and *wait*) in the hybrid automata for each step. These locations synchronize the update of the variables and restore the SFC semantics.

The reachability analysis of SpaceEx returns the states that have been visited. This information useful for the heuristics that compute the refinement of a model. In case the analysis fails due to forbidden states being reached, the tool only returns the states violating the safety condition. Unfortunately, this restriction prevents the output file from showing of the number of visits to each location. In order to calculate this number independent of the SpaceEx result, during the transformation a counting variable is included for each step of the HSFC. This variable is increased if a location corresponding to the step is entered. Additionally, the variables are added to the set of flow variables which do not change their value over time, to ensure a correct interval output. Furthermore, the number of visits are used to notify the user, if certain steps of an HSFC are never visited. In these cases our algorithm warns the user, because any unvisited step may be the result of a restrained SpaceEx configuration.

3.3.1 Transformation to SpaceEx Model and Configuration

During the transformation of the HSFCs, we add additional components to simulate user input and synchronize the variable updates with the PLC cycle, and circumvent the previously described restrictions of SpaceEx to create a model which can be verified by SpaceEx. The transformation of the plant control starts by parsing the SFC XML-File and the conditional ODE XML-File. Both files are combined into a hybrid automaton. During the transformation of an HSFC to an HA , boolean variables are converted to integers and additional counter variables for each step of the HSFC are added as described in Section 3.3. In order to track the total elapsed time, a global timer `global_timer` component is integrated. This global timer component consists of one location and one clock variable, which is never reset.

A controller `control_panel` is also included to simulate random user input. Considering the example tank system in Figure 2.4 in Section 2.3, possible user inputs consist of the pushable buttons controlling the pumps. The `control_panel` component of the model simulates the activation and deactivation of the pumps by setting the variables P_1^{on},P_2^{on},P_1^{off} and P_2^{off}.

After the SpaceEx model file, the configuration file is written. The initial states, scenario, forbidden states and other SpaceEx configurations as introduced in Section 2.8 are defined in this file.

The final SpaceEx model of the component `tank1` without attached conditional ODE systems is illustrated in Figure 3.5. We reduce the transition guards to P_1^{on} and P_1^{off} to create a clearer model.

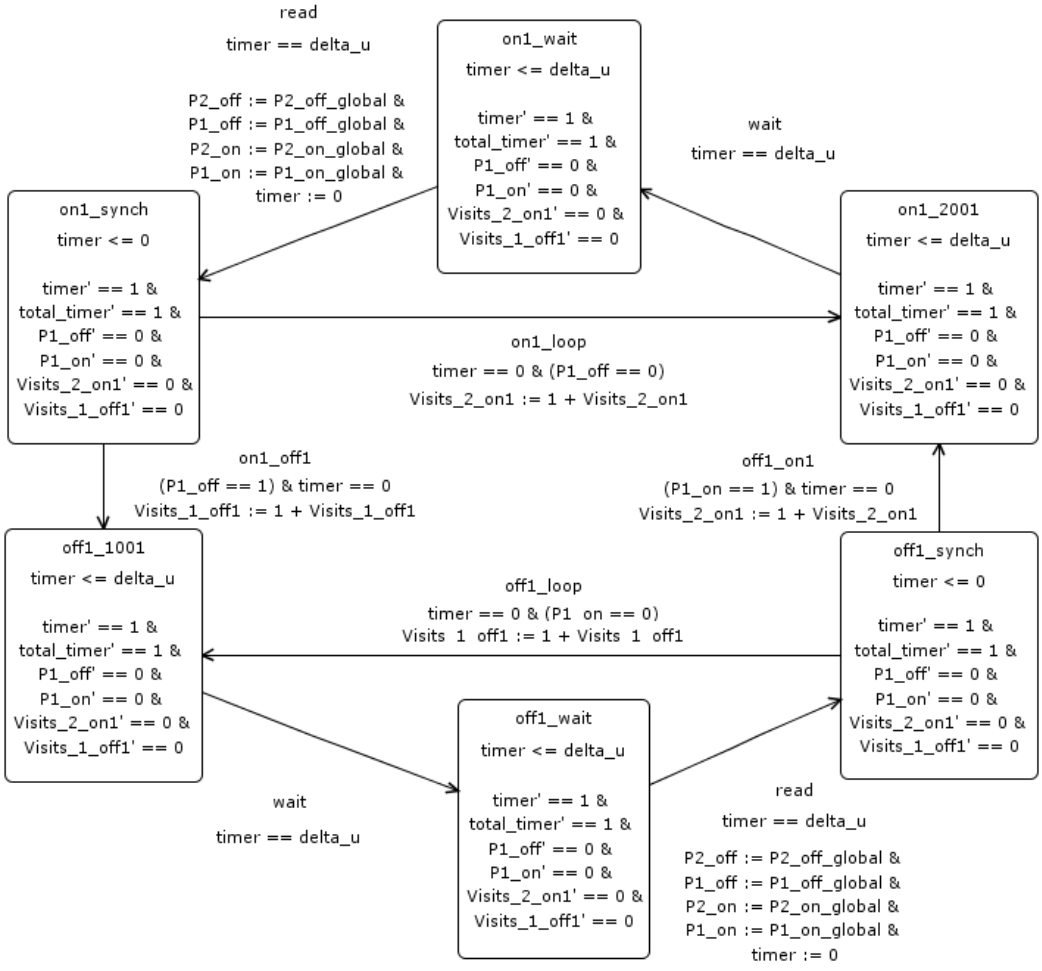

Figure 3.5: *Hybrid automaton of* $tank_1$

3.4 SpaceEx Output

In this section, we present two of the previously mentioned outputs of SpaceEx. The textual output contains the run of a given model and is explained in Section 3.4.1. This sequence of configurations is used to analyze the reachable states. Furthermore, we analyze the interval output which provides the ranges of the variables of the model as described in Section 3.4.2. Both outputs are used during the refinement of the CEGAR-based plant control verification.

3.4.1 Textual Output (TXT-File)

The textual output of SpaceEx provides the run of a model according to the given SpaceEx configurations. When the reachability analysis uses the PHAVer scenario, only the location of each component and the corresponding initial values are displayed. The LGG Support Function scenario also provides the user with the reachable values of the continuous variables in Hessian normal form which allows for the plotting of the resulting polytopes. If the safety verification failed, the run contains only the states where the verification failed.

```
{(loc(control_panel)==control & loc(global_timer)==timer &
  loc(tank1)==off1_1001 & Visits_1_off1 == 1 & Visits_2_on1 == 0 &
  chkb_P1_on == 0 &  P1_on == 0 & ...)|...}
```

Listing 3.3: *Exemplary textual ouput of a PHAVer scenario*

As shown in Listing 3.3, the location of each component of the tank model is given by a function *loc*, which takes a component as an argument and returns its current location. `loc(tank1)==off1_1001` constitutes that the component $tank_1$ is in location `off1_1001`. The values or the ranges of the variables are saved as equations or linear inequalities respectively. Each new configuration is separated by a pipe |.

```
{(loc(tank2)==off2 & loc(tank1)==off1 & ...{[
0,-1,-2,-3,-4,-5,-6,-7,-8,-9,-10
0,-1,-2,-3,-4,-5,-6,-7,-8,-9,-10
0,-1,-2,-3,-4,-5,-6,-7,-8,-9,-10
1, 2, 3, 4, 5, 6, 7, 8, 9,10, 10
1, 2, 3, 4, 5, 6, 7, 8, 9,10, 10
1, 2, 3, 4, 5, 6, 7, 8, 9,10, 10
]}
{variable to dimension map:[timer,tank2.t,tank1.t]
[-1,0,0],[0,-1,0],[0,0,-1],[0,0,1],[0,1,0],[1,0,0]}
<initial_set:tank2.t == -0 & tank2.h1 == -0 & ... & tank1.t <= 10>)|...}
```

Listing 3.4: *Exemplary textual ouput of a LGG Support Function scenario*

As exemplified in Listing 3.4, the LGG Support Function scenario additionally saves the ranges of the continuous variables in Hessian normal form. Every column of values in combination with the subsequent dimension map represents a half-space. The values denote the distance from the origin, while the vectors of the dimension map represent the normal vectors of each plane. By combining these half-spaces a polytope is constructed. For example, the second column in combination with the dimension map generates the polytope shown in Equations (3.2) to (3.7).

$$-1 * timer + \ 0 * (tank2.t) + \ 0 * (tank1.t) \leq -1 \tag{3.2}$$
$$0 * timer + -1 * (tank2.t) + \ 0 * (tank1.t) \leq -1 \tag{3.3}$$
$$0 * timer + \ 0 * (tank2.t) + -1 * (tank1.t) \leq -1 \tag{3.4}$$
$$0 * timer + \ 0 * (tank2.t) + \ 1 * (tank1.t) \leq \ 2 \tag{3.5}$$
$$0 * timer + \ 1 * (tank2.t) + \ 0 * (tank1.t) \leq \ 2 \tag{3.6}$$
$$1 * timer + \ 0 * (tank2.t) + \ 0 * (tank1.t) \leq \ 2 \tag{3.7}$$

The initial set contains the initial values of all variables upon entering the state.

3.4.2 Interval Output (INTV-File)

The interval output can either include all variables of the model or only selected ones. A global variable range is computed for each output variable, which consists of a lower and an upper bound with regard to the complete reachability analysis of SpaceEx. In addition to this global range, location-wise ranges are written into the INTV-File. These ranges provide upper and lower bounds for the variables corresponding to a set of locations. The format of the interval output is independent of the chosen scenario.

```
Bounds on the variables over the entire set:
P1_on: [-0,1]
global_time: [-0,10]
tank2.timer: [-0,10]
V2out_open: [neg_infty,pos_infty]
tank1.Visits_1_off1: [1,5]
...

Location-wise bounds on the variables:

Location: loc(tank2)==off1_1001 & loc(tank1)==off1_1001 & ...
P1_on: [-0,1]
global_time: [-0,10]
tank2.timer: [-0,10]
V2out_open: [neg_infty,pos_infty]
tank1.Visits_1_off1: [1,5]
...
```

```
Location: loc(tank2)==off1_1001 & loc(tank1)==off1_wait & ...
P1_on: [-0,1]
global_time: [10,10]
tank2.timer: [10,10]
V2out_open: [neg_infty,pos_infty]
tank1.Visits_1_off1: [1,3]
...
```

Listing 3.5: *Example of an interval output*

As shown in Listing 3.5, the first set of variables contains the global ranges. For the location-wise ranges, the locations of the components and all variable ranges are given. A bound of `neg_infty` or `pos_infty` denotes that no bound has been found. `V2out_open: [neg_infty,pos_infty]` constitutes that the value of `V2out_open` is between negative infinity and positive infinity.

3.5 Summary

In this chapter, we introduce the CEGAR-based plant control verification by explaining all parts of the analysis. The Beremiz PLC Open Editor is used to model the tank system as an SFC. Furthermore, we present the three parts of the verification input. The Beremiz SFC model and ODE-File are combined to the HSFCs which are transformed into hybrid automata. The guards of the SFC and the conditions of the ODE systems both use a custom condition syntax. Additionally, an XML format to store conditional ODE systems and their associated HSFC steps is proposed. The safety conditions in the form of forbidden states are presented. Afterwards, known restrictions of the tool platform are circumvented during the transformation of the HSFC into a hybrid automaton. Additionally, new components are integrated to monitor the total elapsed time and simulate possible user inputs. The result of the SpaceEx analysis determines, whether a model is safe or the refinement procedure has to be started. A successful SpaceEx verification proves safeness of the given model. If the verification failed for a model and it is not refinable, the model is unsafe. Otherwise, the model is refined. Two of the outputs generated by the SpaceEx analysis are explained in detail. These outputs are used in the refinement as described in the next chapter.

Chapter 4

Counterexample-Guided Abstraction Refinement

In this chapter, we propose an approach to iteratively refine plant models by adding conditional ODE systems. A given model and its SpaceEx output files are used to determine whether the model is safe, unsafe, or unsafe but still refinable. In case the model is still refinable, plant dynamics are used to create a more detailed model by adding conditional ODE systems to the steps of the HSFC. Counterexamples in form of SpaceEx outputs analyzed used to select refinable steps of the HSFC. We employ different heuristics to select the most promising steps for refinement. Figure 4.1 illustrates the refinement procedure, which are applied to refine a given model by considering a counterexample.

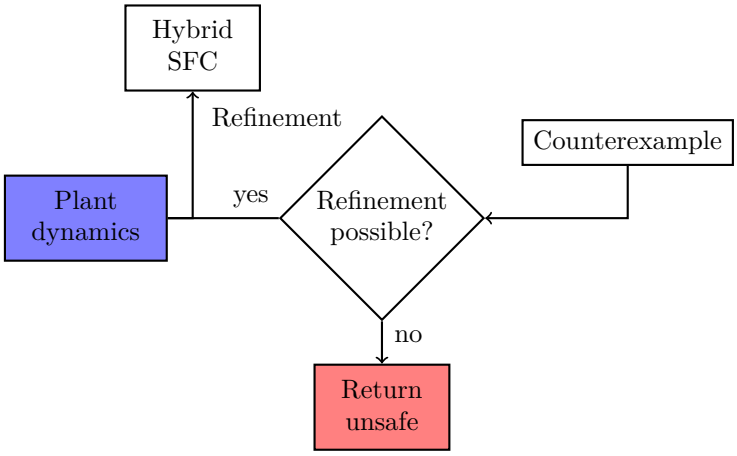

Figure 4.1: *Counterexample-guided refinement*

Firstly, we define the conditions under which a step is refinable by a conditional ODE in Section 4.1. Using this definition we determine whether a model is still refinable during the verification process. This procedure to create a more detailed model are explained in Section 4.2. Afterwards, three strategies to choose a refinable step are presented in Section 4.3. These strategies try to find the most promising refinable steps by employing different heuristics. We discuss how to use SpaceEx outputs to choose the next refinable step(s). The refinement generates are more detailed model of the system by including additional information about the behavior of specific continuous variables. This is accomplished by choosing an appropriate conditional ODE system and joining

it with the step of an HSFC. A description of the step refinement is shown in Section 4.4 together with the explanation how this refinement extends an HSFC and a transformed hybrid automaton.

4.1 Refinable Steps

In this section refinable steps are defined and a procedure is introduced to analyze the SpaceEx output files in order to decide whether a model is refinable. Definition 4.1 formalizes the required conditions for step refinability as follows:

Definition 4.1 (Refinable Step)
A step s of an HSFC C where Var are the variables of C is refinable iff a conditional ODE system $cond\text{ODE}_{sys} = (\text{ODE}_{cond}, \text{ODE}_{list})$ exists where ODE_{cond} denotes the condition and ODE_{list} the set of ODEs, which satisfy the following conditions:

- *$cond\text{ODE}_{sys}$ is not attached to s, i.e., s is not refined by $cond\text{ODE}_{sys}$*
- *ODE_{cond} has to be satisfiable in s, i.e., there exists a variable valuation in at least one location-wise variable range of s, which satisfies ODE_{cond}*
- *ODE_{list} contains at least one ODE with visible variables V, i.e., $V \subseteq Var$*

To check the satisfiability, we use the interval information of a SpaceEx execution. The global variable ranges are not considered as they might include values unreachable in a specific step. A condition ODE_{cond} of an ODE and its ODE system ODE_{list} can be added to a step s, if the condition is satisfiable in the intervals of the condition variables meaning the location-wise variable ranges of s. If there exists a variable valuation in at least one set of these location-wise variable ranges, which satisfies ODE_{cond}, then the condition is satisfied in s. If ODE_{cond} is always false in s, the ODE is not added since its condition will never be satisfied and thus the new location resulting from the ODE, would never be reached. In this case, s is declared not refinable by this ODE, since adding it would have no effect on the reachability analysis.

As an example we analyze, the refinability of step on_1 by the conditional ODE system $condODE_1$ given in Equation (4.1).

$$\overbrace{\underbrace{h_1 \geq 2 \wedge h_1 \leq 8 \wedge P_1 \wedge \neg P_2}_{ODE_{c1}\ (condition)} : \underbrace{\dot{h}_1 = 3, \dot{h}_1 = -3}_{ODE_{l1}\ (list)}}^{condODE_1\ (conditional\ ODE\ system)} \tag{4.1}$$

Considering the interval information presented in Listings 4.1 and 4.2, $condODE_1$ is either added to on_1 or the system $tank_1$ cannot be refined by $condODE_1$. In Listing 4.1, there exists a valuation in a location-wise range which satisfies the condition $condODE_{c1}$. On the other hand, $condODE_{c1}$ is not satisfiable in listing 4.2. The boolean variables P_1 and P_2 are true if they are equal to 1 and false if they are 0. The locations of the hybrid automaton associated with the HSFC step on_1 are named on_{1_x00y} where x and y are internal IDs to identify the locations. The locations with *synch* and *wait* suffixes are used to synchronize (user) input(s) and simulate the PLC cycle. These locations are not considered in the analysis as they may contain values which are unreachable in their corresponding steps.

When examining the satisfiability of a conditional ODE system $condODE_{sys}$, only location-wise bounds are regarded, while bounds over the entire set may include values, which are not reachable in step on_1. Consequently, the condition could be satisfiable over the entire set of locations but not in on_1. Since on_1 is checked for refinability with $condODE_{sys}$, only those intervals associated to on_1 have to be analyzed. Listings 4.1 and 4.2 illustrate two different cases, one where $condODE_1$ is satisfiable and one where $condODE_1$ is not satisfiable.

```
Bounds on the variables over the
    entire set:
tank2.chkb_P2_on: [-0,1]
tank1.chkb_P1_on: [-0,1]
h1: [-0,10]

Location-wise bounds on the variables:

Location: ... & loc(tank1)==on1_2001
tank2.chkb_P2_on: [1,1]
tank1.chkb_P1_on: [1,1]
h1: [-0,10]

Location: ... & loc(tank1)==on1_wait
tank2.chkb_P2_on: [-0,1]
tank1.chkb_P1_on: [1,1]
h1: [-0,10]

Location: ... & loc(tank1)==on1_2001
tank2.chkb_P2_on: [-0,0]
tank1.chkb_P1_on: [1,1]
h1: [-0,10]

Location: ... & loc(tank1)==on1_synch
tank2.chkb_P2_on: [-0,1]
tank1.chkb_P1_on: [-0,1]
h1: [-0,10]
```

Listing 4.1: ODE_{c1} *is satisfiable*

```
Bounds on the variables over the
    entire set:
tank2.chkb_P2_on: [-0,1]
tank1.chkb_P1_on: [-0,1]
h1: [-0,10]

Location-wise bounds on the variables:

Location: ... & loc(tank1)==on1_2001
tank2.chkb_P2_on: [1,1]
tank1.chkb_P1_on: [-0,0]
h1: [-0,1]

Location: ... & loc(tank1)==on1_wait
tank2.chkb_P2_on: [-0,1]
tank1.chkb_P1_on: [1,1]
h1: [-0,10]

Location: ... & loc(tank1)==on1_2001
tank2.chkb_P2_on: [-0,1]
tank1.chkb_P1_on: [1,1]
h1: [-0,1]

Location: ... & loc(tank1)==on1_synch
tank2.chkb_P2_on: [-0,1]
tank1.chkb_P1_on: [-0,1]
h1: [-0,10]
```

Listing 4.2: ODE_{c1} *is* ***not*** *satisfiable*

The variables `chkb_P1_on` and `chkb_P2_on` correspond to P_1 and P_2. Though ODE_{c1} is not satisfied in the first location-wise bounds on the variables of the left interval data (Listing 4.1), since P_2 is always 1 which means it is always true. But ODE_{c1} can be satisfied in the third location-wise bounds. This set of variable ranges includes `chkb_P2_on:[0,0]` thus satisfying $\neg P_2$. Furthermore, `chkb_P1_on:[1,1]` satisfies P_1 and `h1:[-0,10]` provides a valuation of h_1 with $h_1 \geq 2 \wedge h_1 \leq 8$. Therefore $condODE_1$ can refine step on_1 of $tank_1$. Considering Listing 4.2, on_1 cannot be refined by $condODE_1$ because both interval information, where $tank_1$ is in a location associated on_1, do not satisfy ODE_{c1}. The first location-wise interval bounds do not satisfy ODE_{c1} because P_1 is always false (0). ODE_{c1} is not satisfiable in the third location-wise bounds because of $h_1 \in [-0, 1]$ violates $h_1 \geq 2$. The ODE_{c1} could be satisfied in the *synch* and *wait* location(s) of on_1, but since they are used for input synchronization of local and global variables, the intervals may include values unreachable in on_1.

Another restriction on the refinability of steps are based on the equations of $condODE_{sys}$. If all equations of ODE_{list} contain any variables unknown (not visible) to the steps of a system, s cannot be refined by $condODE_{sys}$. If all variables of at least one equation are known, then s can be refined by $condODE_{sys}$, but during the refinement only the equations are added with visible variables. Equations (4.2) and (4.3) show two exemplary conditional ODE systems. Figures 4.2 and 4.3 illustrate which variables are visible to each component and the resulting refinement possibilities.

$$condODE_1 := h_1 \geq 2 \wedge h_1 \leq 8 \wedge P_1 \wedge \neg P_2 : \dot{h}_1 = 3, \dot{h}_2 = -3 \tag{4.2}$$

$$condODE_2 := h_1 \geq 2 \wedge h_1 \leq 8 \wedge P_1 \wedge \neg P_2 : \dot{h}_1 = 3 \tag{4.3}$$

We choose $condODE_2$ as a subsystem of $condODE_1$ to show the effects of ODEs on the refinement.

Variables	$\boldsymbol{tank_1}$	$\boldsymbol{tank_2}$
h_1	✓	
h_2		✓
P_1	✓	✓
P_2	✓	✓

Figure 4.2: *Variables visible to a component are marked with ✓*

ODE	$\boldsymbol{tank_1}$	$\boldsymbol{tank_2}$
$condODE_1$	$\dot{h}_1 = 3$	$\dot{h}_2 = -3$
$condODE_2$	$\dot{h}_1 = 3$	not refinable

Figure 4.3: *Table displaying the ODEs added if a component's step is refined*

System $condODE_1$ contains two equations of which at least one is know to each tank component. The variable h_1 is visible to $tank_1$ while h_2 is not. Thus, only the first equation $\dot{h}_1 = 3$ is added during the refinement. The second equation

$\dot{h}_2 = -3$ is added to steps of $tank_2$ only, since this component knows the variable h_2. The first equation is not added to the $tank_2$, since h_1 is unknown to this component. The second system $condODE_2$ can only be used to refine steps of $tank_1$ as it contains only one equation, which has variables visible to $tank_1$ only. Because $tank_2$ does not know the continuous variable h_1 it does not know any equation of $condODE_2$, no step s of $tank_2$ is refinable by $condODE_2$. If a step is refinable by a conditional ODE system all possible ODEs are added during the refinement.

In the following, two algorithms are implemented to either return a list of all refinable steps (*RefSteps*) for a given conditional ODE system $condODE_{sys}$ or to get all conditional ODE systems $CondSys$, which can refine a given step s. *Components* contains all components of the given model. We define $s \in S_{comp}$ ($s \in S_{comps}$) as the set of steps of the component *comp* (a set of components *comps*). Algorithm 4.1 returns all steps refinable by a given $condODE_{sys}$.

Algorithm 4.1: getRefinableSteps

Data: $condODE_{sys} := (ODE_{cond}, ODE_{list})$
$Comp := Components \backslash \{control\_panel, global\_timer\}$
Result: List of steps *RefSteps* refinable by $condODE_{sys}$

```
RefSteps = ∅;
foreach s ∈ S_Comp do
    if s is not refined by condODE_sys then
        if ODE_cond is satisfiable in s then
            if ∃C ∈ Comp ∧ s ∈ C ∧ vis(C, ODE_list) ≥ 1 then
                RefSteps := RefSteps ∪ {s};
            end
        end
    end
end
return RefSteps;
```

The function $vis(C, \text{ODE}_{list})$ returns the amount of ODEs within ODE_{list}, whose variables are visible to C. The second algorithm *getUnusedODEs* computes a set of conditional ODE systems. Algorithm 4.2 returns a subset of all conditional ODE systems of $CondSys$ containing the conditional ODE systems which can refine a given step s of component C.

Algorithm 4.2: getUnusedODEs

Data: $s \in S_C$
$CondSys := \{condODE_1, \ldots, condODE_n\}$
Result: List of ODEs *RefODEs* which can refine step s

```
RefODEs = ∅;
foreach condODE_sys = (ODE_cond, ODE_list) ∈ CondSys do
    if s is not refined by condODE_sys then
        if ODE_cond is satisfiable in s then
            if vis(C, ODE_list) ≥ 1 then
                RefODEs := RefODEs ∪ {condODE_sys};
            end
        end
    end
end
return RefODEs;
```

4.2 Refinement Procedure

In case the safety verification of the current model has failed, the model is either incorrect or can be further refined. To determine which holds true a *refinement strategy* is executed. This strategy performs two tasks at the same time. The strategy chooses one or multiple refinable steps $s_{ref} \in S_{Components}$, a conditional ODE system $condODE_{ref}$ for each s_{ref} and returns them as tuples. If the model cannot be further refined, the current strategy returns an empty list, thus notifying the verification procedure, that the model is not refinable. Afterwards all s_{ref} in this list are refined by their associated $condODE_{ref}$.

In order to be able to start the refinement procedure after n refinement steps, the current conditional ODE systems *condSys* and their depending steps are stored in a new XML-File. These XML-Files are of the same format as the original XML-File storing *condSys* and can be used to manually start the safety verification based on the current level of refinement. In addition, the current hybrid automaton and the SpaceEx output files are saved to allow for a further analysis of each refinement step.

After the XML-File has been generated, the updated conditional ODE systems and their associated steps are used to create the refined hybrid automaton. Finally, the SpaceEx safety verification for this updated automaton is automatically started as the next iteration.

4.3 Refinement Strategies

The following three *refinement strategies* each aim at computing the refinable steps for the current *Components*. Each strategy employs a different heuristic to determine which refinable steps are chosen in the current iteration of the verification process. The following strategies require input parameters, which are used during the selection of the refinable steps. These parameters are derived from the SpaceEx output files and are used by the refinement strategies.

- Textual output
- Interval global variable ranges
- Interval local-wise variable ranges
- Set of tuples of steps and their visit count (*StepVisits*)
- The conditional ODE systems (*CondSys*) and current refinement
- All components of the model (*Components*)

The textual output provides the reachability analysis of the model as described in Section 3.4.1. The interval global and location-wise ranges variable ranges are explained in Section 3.4.2. The list $StepVisits$ is computed using the interval output before the refinement strategy is executed. This list contains the the number of visits of each step. $ConSys$ contains all conditional ODE systems and $Components$ are all components of the model. The strategies choose one or multiple steps to be refined. If no refinable steps are remaining the strategies return no steps and thus there are no refinable steps and the verification algorithm returns unsafe. The strategy is encapsulated and, consequently, exchangeable to facilitate the development of additional strategies in the future. In the following sections we present two naive refinement strategies (see Section 4.3.1 and Section 4.3.2) and a more elaborated strategy which utilizes the list $StepVisits$.

The strategies can be set to full step refinement meaning chosen steps are refined by all possible conditional ODE systems. This option is not included in the following algorithms but can be achieved by calling $getUnusedODEs$ for a given step. The function returns all conditional ODE systems which can refine this step. Assuming the step s_{ref} is only refinable by $condODE_1$ and $condODE_2$ where $condODE_1, condODE_2 \in CondSys$. If a strategy selects s_{ref} to be refined by $condODE_1$, s_{ref} is additionally refined by all possible conditional ODE systems. In this case s_{ref} is refined by $condODE_1$ and $condODE_2$.

4.3.1 Naive by ODE

This refinement strategy naively selects the first conditional ODE system $condODE_{sys}$ in the conditional ODE systems $CondSys$. The strategy checks whether there are still steps in the $Components$, which can be refined using this ODE. If no refinable steps are left the next $condODE_{sys} \in CondSys$ is selected and checked. This procedure is repeated until a $condODE_{sys}$ is found that can be used for a refinement. The first of the selected steps is chosen for refinement by this naive heuristic. Algorithm 4.3 illustrates the algorithm of this refinment strategy.

Algorithm 4.3: Naive by ODE

Data: $CondSys \coloneqq \{condODE_1, \ldots, condODE_n\}$
Result: $s_{ref} \in RefSteps$ and $condODE_{ref} \in CondSys$

```
foreach condODE_sys ∈ CondSys do
    RefSteps := getRefinableSteps(condODE_sys);
    if |RefSteps| ≥ 1 then
        return {(RefSteps.takeFirst(), condODE_sys)};
    end
end
return ∅;
```

Figure 4.4 lists an exemplary refinement sequence of the *Naive by ODE* strategy under the assumptions that the condition ODE_{cond} of each conditional ODE system $condODE_{sys}$ can be satisfied in each step and at least one ODE is known to the HSFC component of each step and the safety verification of each iteration fails.

	$\boldsymbol{condODE_1}$	$\boldsymbol{condODE_2}$	$\boldsymbol{condODE_3}$
$\boldsymbol{s_1}$	1	4	7
$\boldsymbol{s_2}$	2	5	8
$\boldsymbol{s_3}$	3	6	9

Figure 4.4: *Naive by Ode refinement sequence*

The naive strategy refines all steps sequentially. Assuming there is one component C with $S_C \coloneqq \{s_1, s_2, s_3\}$ and a set of three conditional ODE systems $CondSys \coloneqq \{condODE_1, condODE_2, condODE_3\}$ the strategy starts by selecting the first conditional ODE system $condODE_1$ and refines the first step s_1. In

the next two iterations s_2 followed by s_3 are refined by $condODE_1$. Afterwards, there are no more steps which can be refined by $condODE_1$. Thus, $condODE_2$ is selected to refine the first available step s_1 followed by s_2 and s_3. The same refinement sequence is applied for $condODE_3$. After s_3 is refined by $condODE_3$, no further refinement is possible.

4.3.2 Naive by Step

The second naive strategy as introduced in Algorithm 4.4 takes the first step s of the list of HSFC steps and checks whether a conditional ODE system exists which can refine s. If s has already been refined by all available conditional ODE systems, the algorithm selects the next step. Once the strategy has found a step s_{ref} which can be refined by one or more conditional ODE systems, the first of these systems $condODE_{ref}$ is selected for the current refinement.

Algorithm 4.4: Naive by Step

Data: $Comp := Components \backslash \{control\_panel, global\_timer\}$
Result: $s_{ref} \in Comp$ and $condODE_{ref} \in UnusedODEs$

```
foreach s ∈ Comp do
    UnusedODEs := getUnusedODEs(s);
    if UnusedODEs ≥ 1 then
        return {(s, UnusedODEs.takeFirst())}
    end
end
return ∅
```

Figure 4.5 lists the refinement sequence of the *Naive by Step* strategy, assuming each condition is satisfiable in each step, each step knows at least one equation of each conditional ODE system and the safety verification of each refinement iteration fails.

	$\boldsymbol{condODE_1}$	$\boldsymbol{condODE_2}$	$\boldsymbol{condODE_3}$
$\boldsymbol{s_1}$	1	2	3
$\boldsymbol{s_2}$	4	5	6
$\boldsymbol{s_3}$	7	8	9

Figure 4.5: *Naive by Step refinement sequence*

Assuming there is one component C with $S_C := \{s_1, s_2, s_3\}$ and a set of three conditional ODE systems $CondSys := \{condODE_1, condODE_2, condODE_3\}$, this naive strategy refines step s_1 sequentially by all conditional ODE systems in $CondSys$. In the first iteration s_1 is refined by $condODE_1$. In the two following iterations, s_1 is refined by $condODE_2$ and $condODE_3$. Afterwards, s_1 can not be further refined. The naive strategy chooses the next step s_2 and refines it by $condODE_1$, $condODE_2$ and $condODE_3$ sequentially. The same refinement sequence is applied to s_3 after s_2 is not refinable. After the last refinement iteration where s_3 is refined by $condODE_3$, no further refinement is possible.

4.3.3 First of Most Visited

The third strategy takes a list of steps of the hybrid SFCs and a mapping between steps and their number of visits during the last analysis ($StepVisits$) to select a refinable step and conditional ODE system. This is data is gathered by analyzing the interval output. During the generation of the hybrid automaton, a variable v_s for each step s is added, which is increased by one every time a corresponding location in the hybrid automaton is entered during the reachability analysis. Effectively these variables represent visit counters for the steps. As a result, the global variable range of the interval output contains these variables and their upper bounds which in turn contain the maximum number of visits to the step.

Algorithm 4.5: First of Most Visited

Data: $Comp := Components \backslash \{control\_panel, global\_timer\}$
$StepVisits := \{(s, v_s) | s \in Comp\}$
$CondSys := \{condODE_1, \ldots, condODE_n\}$
Result: $s_{ref} \in RefSteps$ and $condODE_{ref} \in CondSys$

```
while StepVisits ≥ 1 do
    MostVisitedStep := s where max_{v_s}{StepVisits} = (s, v_s);
    foreach condODE_sys ∈ condSys do
        RefSteps := getRefinableSteps(condODE_sys);
        if MostVisitedStep ∈ RefSteps then
            return {(MostVisitedStep, condODE_sys)};
        end
    end
    StepVisits := StepVisits\{(s, v_s)}
end
return ∅
```

Algorithm 4.5 uses the most visited step of the HSFC and checks if it is still refinable by a conditional ODE system. If this is the case, the first unused conditional ODE is added to refine it. If the most visited step is not refinable, the strategy chooses the second, third and so forth most visited step and checks it for refinability.

Assuming ODE_{cond} of each $condODE_{sys} \in CondSys$ are satisfiable in each $s \in S_{Components}$, each ODE_{list} contains at least one equation known to each s, and the safety verification fails in every refinement step. The interval data in Listings 4.3, 4.4, 4.5 and 4.6 represent the SpaceEx output after each iteration. The list of conditional ODE systems contains $CondSys := \{condODE_1, condODE_2\}$. The following listings show four different exemplary interval outputs and refinement cases of a single verification process. The chosen steps are highlighted in green.

```
Bounds on the variables over the entire set:
tank1.Visits_2_on1: [-0,5]
tank1.Visits_1_off1: [1,4]
tank2.Visits_2_on2: [-0,8]
tank2.Visits_1_off2: [1,7]
...
```

Listing 4.3: *Refining on_2 by $condODE_1$*

```
Bounds on the variables over the entire set:
tank1.Visits_2_on1: [-0,3]
tank1.Visits_1_off1: [1,8]
tank2.Visits_2_on2: [-0,5]
tank2.Visits_1_off2: [1,3]
...
```

Listing 4.4: *Refining off_1 by $condODE_1$*

```
Bounds on the variables over the entire set:
tank1.Visits_2_on1: [-0,2]
tank1.Visits_1_off1: [1,9]
tank2.Visits_2_on2: [-0,10]
tank2.Visits_1_off2: [1,3]
...
```

Listing 4.5: *Refining on_2 by $condODE_2$*

```
Bounds on the variables over the entire set:
tank1.Visits_2_on1: [-0,6]
tank1.Visits_1_off1: [1,5]
tank2.Visits_2_on2: [-0,8]
tank2.Visits_1_off2: [1,3]
...
```

Listing 4.6: *Refining on_1 by $condODE_1$*

In the first case (Listing 4.3), the most visited step on_2 of $tank_2$ is refined using the first conditional ODE system $condODE_1$. In the second example (Listing 4.4), the verification of the refined system returns an updated interval output in which off_1 of $tank_1$ is the most visited step and, thus, off_1 of $tank_1$ is

refined by $condODE_1$. In the third iteration (Listing 4.5), on_2 of $tank_2$ is the most visited step again. Since it has already been refined by $condODE_1$, it is refined by $condODE_2$. In the fourth iteration (Listing 4.6), step on_2 of $tank_2$ is still the most visited step, but is no longer refinable, as it has been refined by all $condODE_{sys} \in CondSys$. This causes the strategy to choose the second most visited step, in this case, on_1 of $tank_1$. Consequently on_1 is the refined by $condODE_1$. Figure 4.6 illustrates the refinement sequence of the first four refinement iterations.

	$\boldsymbol{condODE_1}$	$\boldsymbol{condODE_2}$
`on1`	4	...
`off1`	2	...
`on2`	1	3

Figure 4.6: *First of Most Visited refinement sequence*

The strategy can be adapted to count visits only until forbidden states are reached. In this case, the visit count(s) from the textual output, which only calculates the visits after forbidden states have been reached, is subtracted from the number of total visits extracted from the interval output.

4.4 Step Refinement

After a *refinement strategy* has chosen one or multiple pairs of steps s_{ref} and conditional ODE systems $condODE_{ref}$, the $condODE_{ref}$ is attached to s_{ref} in the current HSFC for all chosen tuples (s_{ref},$condODE_{ref}$). During the next generation of the hybrid automaton these new steps are incorporated into the model.

The following subsections consider only the refinement of step $s_m := s_{ref}$, while $m \leq |Steps|$ where $Steps$ are the number of steps in the HSFC C, by one conditional ODE system $condODE_{(n+1)} := condODE_{ref}$. Section 4.4.1 explains the update of the according HSFC during this refinement, while Section 4.4.2 illustrates the changes to the hybrid automaton resulting from the transformation of the HSFC.

4.4.1 HSFC Refinement

The HSFC refinement is accomplished by extending the ODE function Dyn which is introduced in Section 2.5 by the step s_m and the conditional ODE system $condODE_{n+1}$. $condODE_{(n+1)} := (ODE_{c(n+1)}, ODE_{l(n+1)})$ is added to s_m of the HSFC C. Formally, the HSFC of the last iteration of the verification algorithm is extended by $condODE_{(n+1)}$.

$$Dyn_{n+1}(s_i) = \begin{cases} i \neq m : Dyn_n(s_i) \\ i = m : Dyn_n(s_i) \cup \{condODE_{(n+1)}\} \end{cases} \tag{4.4}$$

If Dyn_n is the ODE function of C before refinement and the function shown in Equation (4.4) assigns the conditional ODE systems after C has been refined. After the function Dyn_n has been extended to $Dyn_{(n+1)}$, the refinement of C is complete and the model is transformed into a hybrid automaton during the verification procedure.

4.4.2 HA Refinement

The refinement of s_m by $condODE_{(n+1)}$ creates a new location in the transformed hybrid automaton. Considering $condODE_{(n+1)} := (ODE_{c(n+1)}, ODE_{l(n+1)})$ and the HSFC step s_m, the location in the hybrid automaton is created by adding $ODE_{c(n+1)}$ to the invariant of the location. Furthermore, the appropriate ODEs are integrated as activities into the location. The part of the invariant not resulting from the refinement by $condODE_{(n+1)}$ and the already existing continuous behavior are conjuncted with $ODE_{c(n+1)}$, respectively, $ODE_{l(n+1)}$.

Figures 4.7 and 4.8 illustrate the locations associated with s_m before and after the refinement of s_m by $condODE_{(n+1)}$.

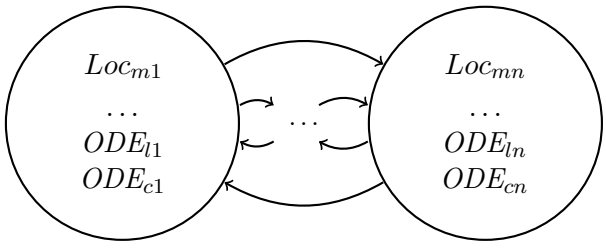

Figure 4.7: *Locations corresponding to s_m before refinement*

The additional location $Loc_{m(n+1)}$, highlighted in red, denotes the location created by $condODE_{(n+1)}$.

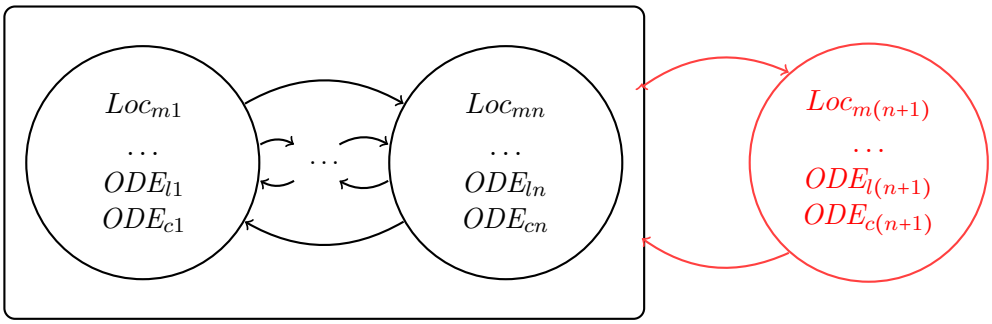

Figure 4.8: *Locations corresponding to s_m after refinement*

If s_m of C was an initial step, the new location in the hybrid system is part of the initial states as well.

We apply the notation for hybrid automata introduced in Definition 2.4. Let $\mathcal{H}_n = (Loc_n, Var_n, Edge_n, Act_n, Inv_n, Init_n)$ be the hybrid automaton of the last iteration. Analogously, the refined hybrid automaton is defined as $\mathcal{H}_{n+1} = (Loc_{n+1}, Var_n, Edge_{n+1}, Act_{n+1}, Inv_{n+1}, Init_{n+1})$. Each location Loc_{ki} is associated with a step s_k of C and the number i with $condODE_i$. The set of variables Var_n stays the same after the refinement. The extended set of locations is shown in in Equation (4.5)

$$Loc_{n+1} := Loc_n \cup \{Loc_{m(n+1)}\} \tag{4.5}$$

To generate the according new edges as stated in Equation (4.6), firstly all transitions are defined that allow the automaton to switch between the conditional ODE locations of s_m. Afterwards, all outgoing transitions to other steps and the self loop created by the HSFC transformation of a step are added. T_m contains all transformed outgoing transitions to other steps and the self loop of s_m as illustrated (see Figure 2.7) in Section 2.6.1.

$$\begin{aligned} Edge_{n+1} := Edge_n &\cup \left\{(Loc_{mi}, \varnothing, Loc_{m(n+1)}) |\ i \in \{1, 2, ..., n\}\right\} \\ &\cup \left\{(Loc_{m(n+1)}, \varnothing, Loc_{mi}) |\ i \in \{1, 2, ..., n\}\right\} \\ &\cup T_m \end{aligned} \tag{4.6}$$

There are no guards and effects on the edges switching between the conditional ODE locations, but the location(s) may only be visited if its invariants are satisfiable. The guards and effects of the step-changing transitions are determined by the transition guards of the HSFC and effects as well as its time guards, which are used to remove Zeno behavior and model the PLC cycle time.

The activities of $Loc_{m(n+1)}$ are extended by the $ODE_{l(n+1)}$ of $Cond_{n+1} : ODE_{n+1}$. Therefore, new behavior of the function Act is added for $Loc_{m(n+1)}$. In this case,

only the ODEs with variables known to C are included. $Equations_{trans}$ contain the equations for the timer t and all variables, which need to be read in the output. The new activities are defined in Equation (4.7) as follows:

$$Act_{n+1}(Loc_{ki}) := \begin{cases} k = m \wedge i = n+1 : Equations_{trans} \cup ODE_{l(n+1)} \\ otherwise \qquad\quad : Act_n(Loc_{ki}) \end{cases} \tag{4.7}$$

The invariant Inv_{trans} created during the transformation from C to the hybrid automaton and $ODE_{c(n+1)}$ are conjuncted. This conjunction denotes the new invariant of location $Loc_{m(n+1)}$. Inv_{trans} contains a guard $(t \leq \delta_u)$ on the timer t, which forces the automaton to take a transition after a specific amount of time. These new invariants are shown in the subsequent Equation (4.8).

$$Inv_{n+1}(Loc_{ki}) := \begin{cases} k = m \wedge i = n+1 : Inv_{trans} \wedge ODE_{c(n+1)} \\ otherwise \qquad\quad : Inv_n(Loc_{ki}) \end{cases} \tag{4.8}$$

The initial states of the hybrid automaton are the locations associated with initial steps of the HSFC combined with the initial valuations of variables V_{init}. If s_m is initial, the initial states of the refined hybrid automaton are extended by the new location. The new set of initial states is defined below in Equation (4.9).

$$Loc_{m(n+1)} := \begin{cases} initial \quad\; : Init_n \cup \{(Loc_{m(n+1)}, v) |\ v \in V_{init}\} \\ otherwise : Init_n \end{cases} \tag{4.9}$$

After all elements of the hybrid automaton have been updated, the automaton has been refined by s_m and $condODE_{(n+1)}$. This new automaton will be checked by SpaceEx in the next iteration.

4.5 Summary

In this chapter the analysis of counterexamples to refine a given plant model is elaborated on. The counterexamples, consisting of SpaceEx tool platform outputs are used to compute HSFC steps which can be refined by conditional ODE systems. A set of conditions for refinable steps is introduced facilitating the selection of applicable conditional ODE systems and steps to be refined. An algorithm is given to evaluate these conditions based on the according SpaceEx output. Three refinement strategies are presented to chose steps which should be refined. These strategies employ different heuristics to select the refinable steps. We presented two naive and one more complex strategy which used the number of visits of each step to compute a refinable step. Afterwards, the refinement of an HSFC step by a conditional ODE system is explained by generating the resulting HSFC and the hybrid automaton. In the following chapter, the iterative CEGAR-based verification procedure presented in this thesis is applied to two examples including the two tank system.

Chapter 5

Experimental Results

In this chapter our verification and refinement approach is applied to two different systems. The example given in Section 5.1 is a reduced tank system with only one pump, while the second example discussed in Section 5.2 represents the example tank system introduced in Section 2.3. Both examples are verified for a set of specific safety properties. An appropriate SpaceEx configuration has to be set, in order to enable the SpaceEx verification to reach all steps of the HSFCs. In case a step is not reached during the SpaceEx analysis, all conditions for this step are assumed to be satisfiable. This might cause the verification to refine steps with conditional ODE systems not satisfiable in the steps.

5.1 One Tank System

We introduce a new, reduced variant of the two tank system (see Figure 2.4) in Section 2.3. This reduced model consists of one water tank only. Additionally, the system includes a pump to drain this tank. The drained water is lost and can not be used to refill the tank. The tank is filled with an initial water level of 50. We employ the safety condition that the tank may never be drained below a water level of $low_1 := 10$. Figure 5.1 illustrates the system.

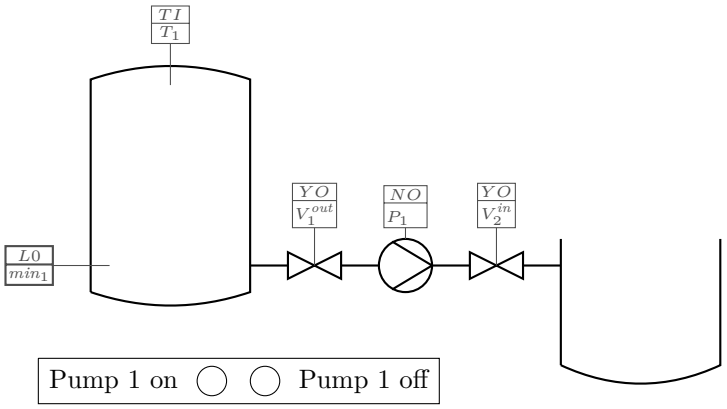

Figure 5.1: *The reduced tank system and its control panel*

We use the SFC illustrated in Figure 5.2 to model this exemplary tank system.

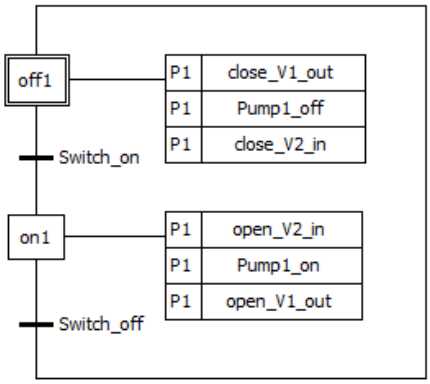

Figure 5.2: *Complete SFC model of the reduced tank system*

The duration t of a PLC cycle is between $\delta_l := 2$ and $\delta_u := 10$ time units meaning $\delta_l \leq t \leq \delta_u$. The transition guards are defined as $Switch_{on} := P_1^{on} \wedge h_1 \geq low_1 + \delta_u$ and $Switch_{off} := P_1^{off} \vee h_1 \leq low_1 + \delta_u$. The SFC is in fact an HSFC without conditional ODE systems and consists of two steps representing the pump states (on_1 and off_1). The conditional ODE systems are defined in the subsequent Listing 5.1.

```
<?xml version="1.0" encoding="UTF-8"?>
<condODEsys>
    <condODE>
        <cond><![CDATA[NOT chkb_P1_on]]></cond>
        <equation>h1' == 0</equation>
    </condODE>
  <condODE>
        <cond><![CDATA[NOT h1 >= min1]]></cond>
        <equation>h1' == 0</equation>
    </condODE>
    <condODE>
        <cond><![CDATA[chkb_P1_on AND h1 >= min1]]></cond>
        <equation>h1' == -1</equation>
    </condODE>
    <init>
        <value var="h1">50</value>
    </init>
</condODEsys>
```

Listing 5.1: *One tank conditional ODE systems*

`chkb_P1_on` stores whether P_1 is currently running or not. Considering that the water level of the tank should never drop below 10, the safety property

which is examined, consists of the following forbidden values. $h_1 < 10$ may never be satisfied in any step of $tank_1$. Three conditional ODE systems are used to refine the initial system. $condODE_1$:= `(NOT chkb_P1_on, h1' = 0)` as well as $condODE_2$:= `(NOT h1 >= min, h1' = 0)` represent the condition $\neg P_1 \vee \neg(h \geq min_1)$ as no disjunction is allowed at the moment. The third system $condODE_3$:= `(chkb_P1_on AND h1 >= min1 , h1' = -1)` models the pump draining water from the tank. As explained in Section 3.3, `NOT h1 >= min1` is transformed into `h1 <= min1`. We apply the *Naive by ODE* refinement strategy as described (see Algorithm 4.3) in Section 4.3.1 with the initial value $h_1 := 50$. Furthermore, we use complete step refinement meaning a chosen step is refined by all possible conditional ODE systems.

A hybrid automaton and a SpaceEx CFG-File are generated for the given model automatically by our verification procedure. During the first iteration, SpaceEx tries to verify $h_1 < 10$ for the SFC. If not specified otherwise, we assume chaotic behavior for the continuous variable h_1. Therefore, the reachability analysis computes a result set, which includes states where $h_1 < 10$ in $tank_1$. Consequently, the SpaceEx verification fails, as shown in Figure 5.3.

```
Execute: ./spaceex_exe/spaceex –config ./resources/1_tank_system/result.cfg -m
./resources/1_tank_system/result.xml -o ./resources/1_tank_system/result.txt -o
./resources/1_tank_system/result.intv -o ./resources/1_tank_system/result.gen -f TXT -f INTV -a "" -a
""
Computing reachable states...
Iteration 0... 1 sym states passed, 3 waiting 0.026s
...
Iteration 49... 50 sym states passed, 16 waiting 0.004s
Performed max. number of iterations (50) without finding fixpoint.
Computing reachable states done after 0.598s
Forbidden states are reachable.
Output of reachable states... 0.311s
```

Figure 5.3: *SpaceEx console output*

First, the *Naive by ODE* strategy selects $condODE_1$. It examines all steps starting with on_1. The maximum range of each location-wise variable range set associated with on_1 is computed. In all sets which correspond to on_1, the range of `chkb_P1_on` is `[1,1]`, thus the condition is not satisfied. The condition `NOT chkb_P1_on` in off_1 is satisfiable as the interval output contains a variable range of `chkb_P1_on:[-0,0]` for off_1. Listing 5.2 illustrates the two condition satisfiability checks.

```
Executing Naive by Ode Strategy...
Cond: chkb_P1_on == 0 - [1.0,1.0] != [0.0,0.0]
UNSAT: chkb_P1_on == 0 - STEP: on1
Cond: chkb_P1_on == 0 - [0.0,0.0] == [0.0,0.0]
SAT: chkb_P1_on == 0 - STEP: off1
Possible Refinement Detected

Refining: tank1.off1 - CondODESystem: chkb_P1_on == 0, Equation=[h1' == 0]
Refining: tank1.off1 - CondODESystem: h1 <= min1, Equation=[h1' == 0]
```

Listing 5.2: *Condition checks of cond*ODE$_1$

By using complete step refinement, all possible conditional ODEs are added to off_1. This means $condODE_2$ also refines off_1. Step off_1 cannot be refined by $condODE_3$ as the condition `chkb_P1_on` is never satisfied in off_1 due to the variable range of `chkb_P1_on:[-0,0]`. Listing 5.3 shows the refinement in the XML-File.

```
<condODE>
    <cond tank1="off1"><![CDATA[NOT chkb_P1_on]]></cond>
    <equation>h1' == 0</equation>
</condODE>
<condODE>
    <cond tank1="off1"><![CDATA[NOT h1 >= min1]]></cond>
    <equation>h1' == 0</equation>
</condODE>
```

Listing 5.3: *First refinement step of the model*

Afterwards, a new hybrid automaton is generated based on the updated HSFC. In the second iteration, the safety verification also fails due to step on_1, which still exhibits chaotic behavior for the continuous variable h_1. This chaotic behavior causes the SpaceEx verification to fail. The *Naive by ODE* refinement strategy is executed a second time to select the next step and conditional ODE system.

Again, the first conditional ODE system $condODE_1$ is selected. As off_1 has already been refined by $condODE_1$, only step on_1 is examined. In this case, the condition `NOT chkb_P1_on` of $condODE_1$ is not satisfiable, because the range of `chkb_P1_on:[1,1]` in every occurrence of on_1 in the interval output. Thus, no further steps are refinable by $condODE_1$, which causes the *Naive by ODE* strategy to continue with $condODE_2$. Step off_1 has already been refined by $condODE_2$. Step on_1, however, can be refined by $condODE_2$ as all requirements defined in Definition 4.1 are met. The continuous variable h_1 exhibits chaotic behavior in on_1, thus `NOT h1 >= 2` is satisfiable. Listing 5.4 shows the condition checks of the second iteration.

```
Executing Naive by Ode Strategy...
Cond: h1 <= min1 - [-Infinity,Infinity] <= [10.0,10.0]
SAT: h1 <= min1 - STEP: on1
Possible Refinement Detected

Refining: tank1.on1 - CondODESystem: h1 <= min1, Equation=[h1' == -1]
Refining: tank1.on1 - CondODESystem: chkb_P1_on == 1, Equation=[h1' == -1]
```

Listing 5.4: *Condition checks of cond*ODE$_1$

`[-Infinity,Infinity] <= [10.0,10.0]` is satisfiable as there exists a value $v \in [-\infty, \infty] \wedge v \leq 10$. All possible conditional ODE systems refine on_1 due to the complete step refinement. The chaotic behavior of h_1 and the variable range of `chkb_P1_on:[1,1]` in on_1, cause the step to be refinable by $condODE_3$. The refinement of on_1 by $condODE_2$ and $condODE_3$ is given in Listing 5.5

```
<condODE>
    <cond tank1="off1,on1"><![CDATA[NOT h1 >= min1]]></cond>
    <equation>h1' == 0</equation>
</condODE>
<condODE>
    <cond tank1="on1"><![CDATA[chkb_P1_on]]></cond>
    <equation>h1' == -1</equation>
</condODE>
```

Listing 5.5: *Second refinement step of the model*

After the model has been refined by $condODE_2$ and $condODE_3$, h_1 exhibits no chaotic behavior as we do not consider the default location in this example. Starting with $h_1 := 50$, the value remains unchanged in off_1 or is decreased in on_1, if it is larger than $low_1 + \delta_u$ based on the refinement of the steps. Adding $condODE_3$ to on_1 models unwanted behavior in the step, but this behavior is excluded by the forbidden states. If the SpaceEx analysis reaches on_1 and the condition `NOT h >= 10` of $condODE_3$ is satisfied, the analysis fails due to forbidden states ($h_1 < 10$) being reached.

SpaceEx now verifies the safety condition $h_1 \geq 10$ for the HSFC model in Figure 5.4. The value h_1 does not drop below 10 as only on_1 decreases h_1. The model can stay in on_1 if $h_1 \geq (low_1 + \delta_u)$. Otherwise the transition $Switch_{off}$ is enabled and taken due to the urgency of SFC transitions. Even if on_1 is entered at a water level of 20 and stays a full PLC cycle, the water level can not drop below 10 as the maximum cycle time is $\delta_u = 10$. Consequently, the continuous variable h_1 reaches its minimum value of $20 - (1 * \delta_u) = 10$. Thus, staying in on_1 and lowering the value of h_1 indefinitely is impossible.

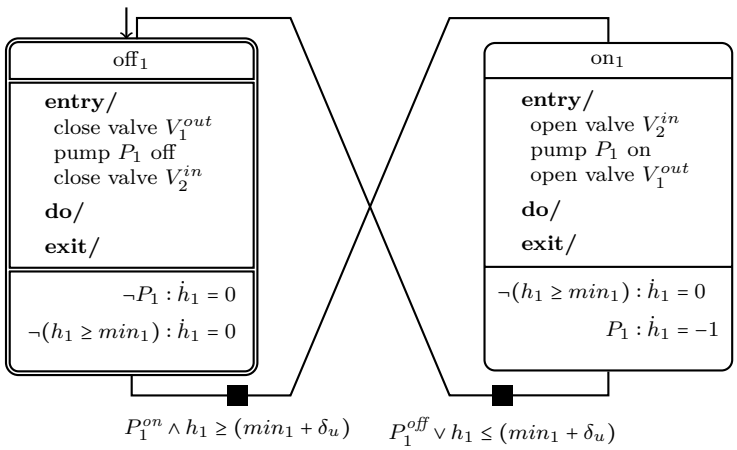

Figure 5.4: *Safe HSFC of $tank_1$*

The model is labeled as *safe*. The average runtime of SpaceEx for each iteration did not change distinguishably. In each iteration the verification took approximately 1.835 seconds, while the complete verification procedure took approximately 9.243 seconds. Listing 5.6 shows the result of the last SpaceEx analysis and of the CEGAR-based verification.

```
...
Iteration 199... 200 sym states passed, 37 waiting 0.007s
Performed max. number of iterations (200) without finding fixpoint.
Computing reachable states done after 2.513s
Forbidden states are not reachable.
...
SpaceEx analysis is finished!
-----------------------------
THE MODEL IS CORRECT!
Number of Refinement Steps: 2
Refinement Sequence:
Step: off1 | condODE: Cond= chkb_P1_on == 0, Eqn=[h1' ==  0]
Step: off1 | condODE: Cond= h1 <= min1     , Eqn=[h1' ==  0]
Step: on1  | condODE: Cond= h1 <= min1     , Eqn=[h1' ==  0]
Step: on1  | condODE: Cond= chkb_P1_on == 1, Eqn=[h1' == -1]
Computation completed after: 11.4688s
```

Listing 5.6: *Result of the CEGAR-based verification*

The SpaceEx analysis does not reach forbidden states. Consequently the model is safe. The number of refinement steps and the refinement sequence are displayed. Additionally, the total computation time of the CEGAR-based approach is given. Figure 5.5 illustrates the reachable water levels `h1` over time `global_time`.

A previous version example, which did not include the correct transition guards and a different set of conditional ODE systems, revealed a problem of the current verification. on_1 refined by $h_1 \geq 10 \wedge P_1 : \dot{h}_1 = -1$ could be reached, but

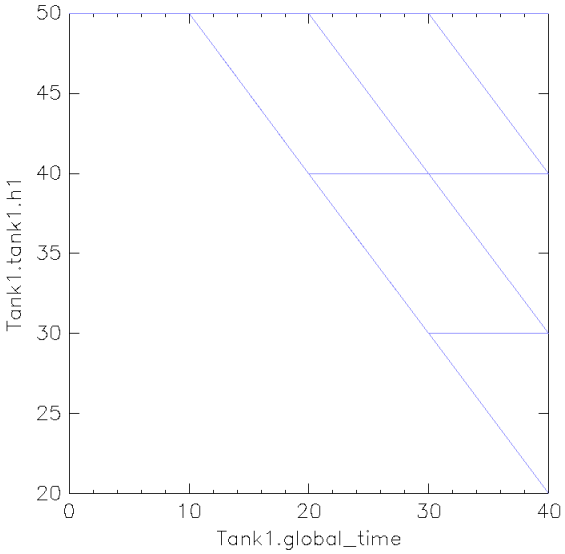

Figure 5.5: *Reachable water levels* `h1` *over time* `global_time`

the water level is not dropping below 10 as there is no location corresponding to step on_1 where $h_1 \geq 10 \wedge P_1$ does not have to be satisfied. Consequently, on_1 can not be left and no timed step can be done as this would violate the location invariant. Omitting the default location (see Section 2.6.2) causes the algorithm to consider systems safe, which are not. If no condition of a conditional ODE system is satisfied in a step, we have to assume chaotic behavior. Since at the moment we are not able to model this default location, we can not reproduce this behavior.

To avoid this problem we use conditional ODE systems with conditions that are complete, i.e., whose disjunction evaluates to *true*. Additionally, we apply the complete step refinement. These two conditions allow us to omit the default state in this example.

5.2 Two Tank System

This system regarded in this section is identical to the tank system which has been described in Section 2.3. The maximum water level of each tank is 50. The variables h_1 and h_2 denote the water levels in $tank_1$ and $tank_2$. The initial values of h_1 and h_2 are $h_1 := 25$ and $h_2 := 25$. In the following, we try to verify the model for the forbidden states $h_1 < 10 \vee h_1 > 40 \vee h_2 < 10 \vee h_2 > 40$ because we do not want either tank to overflow or to be drained completely. The SFCs presented in Section 3.2.1 and the conditional ODE systems given in Equations (2.11) to (2.14) model our tank system. The sensors min_1 and max_1 (min_2 and max_2) are used to detect the values larger than 10 and 40 respectively. The corresponding XML-File of the conditional ODE systems can be seen in Listing 3.1. Unfortunately, this model is not a sufficient model for the tank system as either tank can overflow. As a consequence, the given model is incorrect.

For example, if during a PLC cycle while P_1 is running max_2 detects a high water level in T_2, P_1 is turned off during the next cycle because the T_1 gets the updated value of max_2 not until the beginning of this next cycle. During this second cycle, T_2 assumes, that P_1 is still running as P_1 has not yet been turned off. P_1 is turned off in this cycle. At the beginning of the third cycle T_2 finally receives the updated value of P_1, thus the water flow, which is filling T_2, is stopped. During the first two cycles, the water level h_2 of T_2 can exceed 40.

In this second example, we apply the *First of Most Visited* refinement strategy as elaborated on in Section 4.3.3, which selects the most visited refinable step in each iteration. Since the model is incorrect, our verification procedure does not stop until all meaningful steps are refined. Additionally, we employ complete step refinement.

The complete refinement sequence and added ODEs (highlighted in green) are illustrated in Figure 5.6. In the first iteration, all continuous variables exhibit chaotic behavior. Thus, the verification fails because both h_1 and h_2 can reach values below 10, respectively, above 40. The most visited step off_1 of $tank_1$ is refined by the all possible conditional ODE system. In the second iteration, step off_2 is the most visited step and the algorithm continues refining it. Afterwards, step on_1 is the most visited refinable step. There exist two satisfiable conditional ODE systems which are attached to the step until it is no longer refinable. Finally, the steps on_1 and on_2 are refined.

Iteration	Step	Conditional ODE System
1	off_1	$\neg P_1 \wedge \neg P_2 : \dot{h}_1 = 0, \dot{h}_2 = 0$
	off_1	$\neg P_1 \wedge P_2 : \dot{h}_1 = 2 , \dot{h}_2 = -2$
2	off_2	$\neg P_1 \wedge \neg P_2 : \dot{h}_1 = 0 , \dot{h}_2 = 0$
	off_2	$P_1 \wedge \neg P_2 : \dot{h}_1 = -1 , \dot{h}_2 = 1$
3	on_1	$P_1 \wedge \neg P_2 : \dot{h}_1 = -1 , \dot{h}_2 = 1$
	on_1	$P_1 \wedge P_2 : \dot{h}_1 = 1 , \dot{h}_2 = -1$
4	on_2	$\neg P_1 \wedge P_2 : \dot{h}_1 = 2 , \dot{h}_2 = -2$
	on_2	$P_1 \wedge P_2 : \dot{h}_1 = 1 , \dot{h}_2 = -1$

Figure 5.6: *Refinement sequence (FirstOnMostVisited)*

Further refinement is impossible because in the steps off_1/off_2 (on_1/on_2), P_1/P_2 ($\neg P_1/\neg P_2$) are never satisfied. For example off_1 cannot be refined by $P_1 \wedge P_2 : \dot{h}_1 = 1, \dot{h}_2 = -1$ as P_1 is not satisfiable in off_1.

After conducting all possible refinements of the current model, the SpaceEx verification computes the set of reachable states, which still includes forbidden states. These forbidden states are reachable because a pump might still be assumed running by another tank. This can cause a tank to overflowing as described previously. Our verification procedure evaluates the refinability of the model, but since no additional refinements are possible and the SpaceEx verification fails, the model is labeled *unsafe* as shown in Listing 5.7.

```
...
Forbidden states are reachable.
Output of reachable states... 6.255s
SpaceEx Verification failed!
SpaceEx analysis is finished!
----------------------------
Executing First On Most Visited Strategy...
...
THE MODEL IS INCORRECT!
Number of Refinement Steps: 8
No more refinable steps available!
Refinement Sequence:
Step: off1 | condODE: Cond= chkb_P1_on == 0 & chkb_P2_on == 0, Eqn=[h1' ==  0]
Step: off1 | condODE: Cond= chkb_P1_on == 0 & chkb_P2_on == 1, Eqn=[h1' ==  2]
Step: off2 | condODE: Cond= chkb_P1_on == 0 & chkb_P2_on == 0, Eqn=[h2' ==  0]
Step: off2 | condODE: Cond= chkb_P1_on == 1 & chkb_P2_on == 0, Eqn=[h2' ==  1]
Step: on1  | condODE: Cond= chkb_P1_on == 1 & chkb_P2_on == 0, Eqn=[h1' == -1]
Step: on1  | condODE: Cond= chkb_P1_on == 1 & chkb_P2_on == 1, Eqn=[h1' ==  1]
Step: on2  | condODE: Cond= chkb_P1_on == 0 & chkb_P2_on == 1, Eqn=[h2' == -2]
Step: on2  | condODE: Cond= chkb_P1_on == 1 & chkb_P2_on == 1, Eqn=[h2' == -1]
Computation completed after: 2119.1602s
```

Listing 5.7: *Result of the CEGAR-based verification*

In each iteration the verification took approximately 240.653 seconds, while the complete verification procedure took approximately 2180.273 seconds. These large iteration times are due to the extended number of iterations set for the SpaceEx analysis. If the number of iterations is too low, not all steps are reached. Evidently, the model is not adequate as the delayed variable updates due to the PLC cycles cause the water level changes of the tanks to be asynchronous.

5.3 Summary

In this chapter we exemplified two different variants of the verification and refinement process. In the first example a reduced one pump system is used, where the tank is drainable by a pump. We apply the naive refinement strategy *Naive by ODE* to the first example. Our verification procedure needs three iteration to verify the one tank system for a given safety property. The model for this example is correct. Unfortunately, a previous version of the tank model revealed a problem of our algorithm, which causes the verification procedure to falsely verify a tank model. We are currently working on a solution for this problem. The second example is the two tank system which has been introduced in the previous chapters. As *Naive by Step* is a naive strategy as well, we apply the more complex strategy *First of Most Visited*. The model of the two tank system is not sufficient. The safety property is violated even when no further refinements are possible. In both cases our verification procedure computes the correct result.

Chapter 6

Conclusion and Future Work

The proposed iterative CEGAR-based verification approach provides an automated process to analyze plant control. The initial input SFCs and plant dynamics are combined into HSFCs. Afterwards, this HSFCs is transformed into HA. Safety condition and the third-party tool SpaceEx are used to analyze the HA. A successful SpaceEx verification proves the model safe. If the analysis fails, our verification procedure checks if the current model is still refinable. A refinement strategy is executed to select one or multiple refinable steps and conditional ODE systems. These steps are refined by the conditional ODE systems. Subsequently, the new model is analyzed in the next iteration of the verification. This verification cycle is performed until either SpaceEx verifies the model or it cannot be refined further. In case no refinement is possible and the SpaceEx verification still fails, the model is unsafe.

The discrete and continuous behavior of a plant are verified, because the method integrates control logic and the plants dynamic behavior into the analysis. This type of system analysis may generate a smaller, verifiable model, thus circumventing a state-space explosion which might occur. We identified the number of refinement steps and the configuration of the hybrid automata verification tool to be important factors referring to the runtime of the algorithm. Verifying small examples, i.e., example with few steps and conditional ODE systems, does not exhibit distinguishable changes in the runtime of each iteration. In these cases using the complete model as the initial model should be preferred as only one safety verification has to be performed. Since a large model can result in a state-space explosion the CEGAR-based verification approach should be favored. We need additional and larger models to get conclusive results about which approach is recommendable for different models.

The experimental results revealed a problem of the current algorithm. An unsafe may be declared safe due to missing chaotic behavior. In case no conditional ODE systems are satisfied in a step, a default location has to be entered where the continuous variables exhibit chaotic behavior. This default location has not yet been included in the verification. To model the behavior of the plant correctly this location will implemented as future work.

Based on a modular architecture, exchangeable refinement strategies allow for convenient modifications of the refinement process and adaptation with regard to different inputs. The number of refinement steps might vary depending on the applied strategy. The preliminary results are promising but the verification

of more advanced models is required before conclusive statements about strategy efficiency may be made. In conclusion the our verification approach constitutes an intelligible and convenient technique to verify plant controls and dynamics.

There are several aspects concerning our verification procedure, which can be extended in the future. These extensions will allow the user to analyze more complex plants and use additional functionalities. New strategies and transformations could be implemented to ensure the compatibility with other verification tools for hybrid automata. In addition, the already existing strategies can be adapted to support these other tools as well. By using different safety verification tools, the performance of the verification and refinement might change. This would allow for further test scenarios based on various combinations of strategies and verification tools in order to determine the fastest combination to verify plants.

The presented refinement strategies can be improved by allowing the user to specify the number of steps which are refined during one iteration. Instead of one refinable step, each strategy would select several and refine them in same refinement iteration. This feature would affect the runtime of our verification approach as fewer reachability analyses have to be performed. Furthermore, it is possible to adapt the more elaborated strategy to compute the steps which were visited last, before the verification failed. Refining these steps may result in a faster verification of a model.

Currently, our approach supports only three different action qualifiers of SFCs. The transformation of the SFC and of the plant dynamics is suited to transform additional qualifiers into a hybrid automaton as well. For example, a time delayed action, activating some time after having entered the associated step might be integrated into the transformation.

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 15, 17, 19 and 20)

[BCMP98] L. Baresi, S. Carmeli, A. Monti, and M. Pezzé:. PLC Programming Languages: A Formal Approach. Associazione Nazionale Italiana Per L'Automazione, 1998.
(cited on page 1)

[CFL10] S. Cotton, G. Frehse, and O. Lebeltel. The SpaceEx Modeling Language. `http://spaceex.imag.fr/sites/default/files/spaceex_modeling_language_0.pdf`, 2010.
(cited on page 24)

[ELS05] S. Engell, S. Lohmann, and O. Stursberg. Verification of embedded supervisory controllers considering hybrid plant dynamics. *International Journal of Software Engineering and Knowledge Engineering*, 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, *Proceedings of the 23rd International Conference on Computer Aided Verification*, 2011.
(cited on pages 2, 20, 24 and 29)

[Fre05] G. Frehse. PHAVer: Algorithmic Verification of Hybrid Systems Past HyTech. In *Proceedings of the 8th International Conference on Hybrid Systems: Computation and Control*, 2005.
(cited on page 20)

[GG09] C. Guernic and A. 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 20)

[HKD98] G. Hassapis, I. Kotini, and Z. Doulgeri. Validation of a SFC Software Specification by using Hybrid Automata. In *Proceedings of the 9th Symposium on Information Control in Manufacturing*, 1998.
(cited on page 1)

[IEC03] *An Open Source IEC 61131-3 Integrated Development Environment*, 2003.
(cited on pages 6, 25)

[JP00] M. Joswig and K. Polthier. JavaView JVX Format. `http://www.eg-models.de/formats/Format_Jvx.html`, 2000.
(cited on page 21)

[MT00] R. Maier and Nick Tufillaro. Gnu Plotutils. `http://www.gnu.org/software/plotutils/`, 2000.
(cited on page 21)

[NA12] J. Nellen and E. Ábrahám. Hybrid Sequential Function Charts. In *Proceedings of the 15. Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, 2012.
(cited on pages 1, 10, 14, 15 and 19)

[Pol06] K. Polthier. Javaview. `http://www.javaview.de`, 2006.
(cited on page 21)

[Tis12] E. Tisserant. Beremiz PLC Open Editor. `http://www.beremiz.org/`, 2012.
(cited on pages 2, 24 and 25)