hybr2d Theory of Hybrid Systems Informatik 2
RWTH AACHEN

Bachelor of Science Thesis

Datatypes and tools for the analysis of hybrid systems

Kim Maren Haps

Supervisors:
Prof. Dr. Erika Ábrahám

Advisor:
Johanna Nellen
Xin Chen

06.06.2013

Erklärung

Hiermit versichere ich, dass ich die vorgelegte Arbeit selbstständig verfasst und noch nicht anderweitig zu Prüfungszwecken vorgelegt habe. Alle benutzten Quellen und Hilfsmittel sind angegeben, wörtliche und sinngemäße Zitate wurden als solche gekennzeichnet.

Kim Maren Haps
Aachen, den 06. Juni 2013

Acknowledgements

Writing the Bachelor Thesis was only possible with the help, time and effort of many people. Hereby, I would like to express my appreciation to all the people who supported my research and helped me with the creation of this thesis.

First of all, I would like to pass the gratitude to my supervisor Erika Ábrahám and my advisors Xin Chen and especially Johanna Nellen. Thanks for their great supervision and patience. I could not have finished this thesis without their valuable advice.

Secondly, I would like to thank the university, RWTH Aachen. Thanks for offering me the opportunity to study here, as well as grateful gratitude to all the teachers and nice classmates.

Last but not least, I am deeply grateful to my parents who have been encouraging and motivating me throughout my study in Aachen. Without their support spiritually and financially, I could not have finished my study.

Thank you sincerely!

Kim Maren Haps

Contents

Chapter 1

Introduction

In industry the use of plants in production processes is very common. For the automation of the plants it is resorted to *programmable logic controllers* (PLCs). They control the behavior the plants. To program such a PLC the industry standard IEC 61131-3 [1] offers several languages.

The *sequential function charts* (SFCs) are one of them. This language describes the behavior of a process in a graphical way. It allows to split the process into steps, which is very helpful, when building and analyzing large and complex system.

A wrong programmed SFCs can lead to damages at the plant, for instance a pump of the plant can run dry and get broken, if there is no safety mechanism to prevent this. To detect such unwanted behavior that could occur during a run of PLC a previous verification of the SFCs is recommended.

The SFC verification tool offers the opportunity to check SFCs for safety. Since the SFCs only specifies the control of a plant, the behavior of the underlying plant must be determined as well to get a full safety check. This behavior can be described by conditional ODE systems. They specify under which conditions some component of the plant has the defined behavior. The conditional ODE systems are assigned to some steps to combine the control of the plant given by the SFCs with the behavior, that occurs during the execution of the program. The resulting system is called a *hybrid sequential function chart* (HSFC) [2]. Since there is no approach, which is able to check HSFCs directly, the SFC verification tool takes the HSFCs and transforms them into hybrid automata. These automata are given to SpaceEx, an analysis tool for hybrid automata [3]. The analysis of hybrid automata is very time intensive, so the SFC verification tool tries to keep the model as small as possible. The conditional ODE systems are not fully assigned to the SFCs at the beginning, but are added stepwise during the CEGAR-based verification process. The first analysis of the automata is done on the SFCs without any conditional ODE system. The result of the executed analysis, which is safe or unsafe. In the latter case, it gives back a counter example, which is used to determine a suitable candidate of the given ODE systems to be assigned to the SFCs. So an additional ODE systems is added during a refinement step. The verification is done when the model is safe or there are no ODE systems to add, so the model is unsafe [4].

The transformation from the SFCs respectively the HSFCs to hybrid automata is customized to support the SpaceEx analysis tool. All restrictions, which are given by SpaceEx such as supporting linear convex conditions only, are applied to the automata while building them. The SFC verification tool directly writes the automata to the input file for SpaceEx, there is no data structure built for the automata. This restricts our tool to use of only SpaceEx.

While working on the project, we discovered some bugs in SpaceEx that prevent a proper analysis of our SFCs. Therefore we plan to integrate other analysis tools to the SFC verification tool. This will also increase the flexibility.

We implemented a new data structure for hybrid automata and adapted the transformation from (H)SFCs into hybrid automata so that it uses this data structure instead of creating the SpaceEx input file directly. The new transformation is independent from any restriction given by SpaceEx and any other analysis tool. The resulting hybrid automata are saved in our newly implemented data structure. If we want to make the automata suitable for the analysis by a specific tool, we can apply some other transformations on the data structure afterwards. These transformations can be done by a *ToolChain*, which can handle the restrictions of different tools.

Beside the data structure we introduced a component, which builds the parallel composition of the hybrid automata of a system. Previously this was not needed, because SpaceEx can compute the parallel composition of a set of hybrid automata on the fly during the reachability analysis. In some test cases we found out, that there are bugs in the computation of the parallel composition. This is one reason for implementing the building of the composition as an own component in our tool. The other reason is, that there are tool, which can not handle the analysis for several automata. So if we want to use them for the computation of the reachable states, we must provide them the parallel composition of our set of hybrid automata. With these improvements our tool is now as flexible as possible for the integration of different analysis tools.

The thesis is structured as follows: We introduce an example plant system, sequential function charts respectively hybrid sequential function charts, hybrid automata and their parallel composition as well as the SFC verification tool and its transformation from (H)SFCs to hybrid automata in chapter 2. In chapter 3 we present the data structure for the hybrid automata. The building of the composition by our tool is explained in chapter 4. The description of two benchmarks and the comparison of our parallel composition with the composition SpaceEx computes can be found in chapter 5. We finish the thesis with with a conclusion and the future work in chapter 6.

Chapter 2

Preliminaries

In this chapter we introduce an example plant that we use in this thesis. Afterwards we explain SFCs, hybrid SFCs, and hybrid automata. In the end we describe the SFC verification tool including the transformation from (H)SFCs to hybrid automata for SpaceEx. This basics are needed for the understanding of the main part of this thesis.

2.1 Plants

Since the SFC verification tool is used for checking of chemical plants, we give a simple example of it, that we in this thesis as a running example. This example is retrieved from [2] and depicted in figure 2.1.

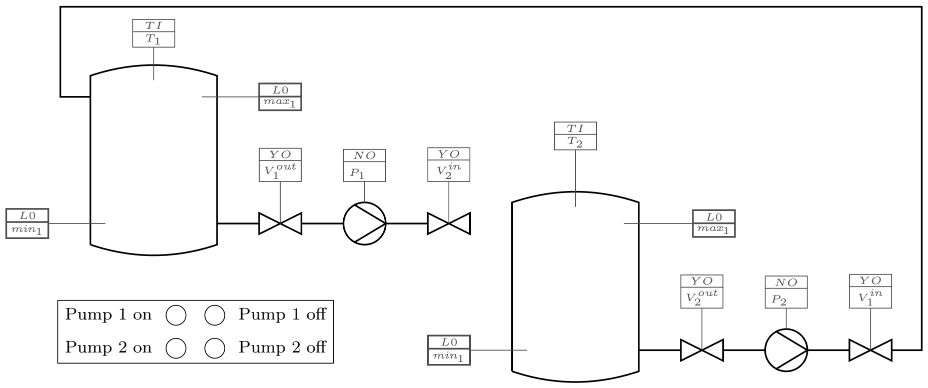


Figure 2.1: An example plant and its control panel from [2]

In this plant we have two equally build tanks T_1 and T_2, which are filled with water. The level of each tank T_i is denoted by the variable h_i. To detect a critical water level, each has tank has two sensors. Near the bottom of the tank is a sensor, which signals a low water level ($\neg min_i$), if it is not covered by water. In this case the tank T_i is starting to run dry, if more water leaves the tank. The other sensor is near the top of the tank. When it gets in contact with water, it detects a high water level (max_i), which can cause the tank to flood. We have a wanted water level, if the two sensors signal ($min_i \wedge \neg max_i$).

Beside the tanks, we have two pumps P_1 and P_2 in the plant. They are connected each by pipes with both tanks. The pumps can be switched on or off manually by using the control panel. This is indicated by the variables (P_i) respectively ($\neg P_i$). If a pump P_1 is running, it pipes an amount c_1 of water per time unit from tank T_1 into the other tank. The water levels of the tanks lower respectively raise by c_1 while the pump is turned on. This holds vice versa for pump P_2. It pumps the water from tank T_2 back into tank T_1 with a capacity of c_2.

A control program is aware of the water levels of the tank to prevent a pump from running dry or the tanks from flooding. It observes the tanks with the sensors. The command to switch a pump on is only executed if the water level of the tank, from which the water piped out, is not low, and the water level of the other tank is not high.

2.2 Sequential function charts

To model the control for a plant we use the *sequential function charts (SFCs)*. They offer us the possibility of a graphical representation of the control and allow us to split the control sequences into steps. The following description holds only for a restricted set of sequential function charts (*SFC*) defined by the IEC standard 61131-3 [1]. The differences are explained, when they occur in the description.

A sequential function chart (*SFC*) has a set of variables *Var*. There are three different types of variables: input, output and local variables. Input variables are set by the environment or another SFC. Output and local variables can be changed by the SFC during execution. But while output variables can be read from the outside of an SFC, local variables are just for the internal use of an SFC. Each variable has a data type such as integer, real or boolean. The IEC standard 61131-3 offers more data types[1].

Mainly the structure of an SFC is determined by set of steps *Steps* and a set of guarded transitions *Trans*. The execution starts in an initial step s_0, which is at the top of the SFC. All other steps are normally arranged top down accordingly to the order they should be executed in. A transition of an SFC connects two steps with each other. It leads from the bottom of the source step to the top of the target step, thus it is directed.

When entering a step, it becomes activated and the set of action blocks *Blocks*, which are assigned to the step, are performed. The set of all action blocks in an SFC is *B*. Every action block consists of an action qualifier and an action itself. An action qualifier determines, when the corresponding action should be performed. There are three types of it: *entry*, *do* and *exit*. The action qualifier *entry* causes that an action is executed only once, when entering a step. So does the qualifier *do*, but its action is also performed the whole time, when the step is activated. When the step is deactivated, thus when it is left via a transition, the action of the qualifier *exit* is executed [2]. In the IEC standard *P1, N* and *P0* are used for the action qualifiers. For a better understanding we renamed them with enter, do and *exit.* The IEC standard also defines more of these qualifiers, which are not considered here [1].

There can be more than one action block with the same action qualifier assigned to a step. In this case the action are performed in a sequence formally given by a total order $\sqsubset$. In the graphical representation the order is depicted by arranging the actions top down. In figure 2.2, for example, there are two actions $action_2$ and $action_3$ with a *do* qualifier. According to their arrangement in the step, $action_2$ would be executed before $action_3$.

There two types of actions. It can either change the value of a variable by a variable assignment or it executes another SFC, which is attached to it. Here the history flag *Hist* becomes important. If this flag is set to *false*, the execution of the nested SFC is done by activating the initial step. If the history flag is set to *true*, the last active step of this SFC is reactivated. In the graphical representation of an SFC an action is denoted by an expressive name for what it does. For instance, if the action is $Pump_{on} = 1$, the action name could be *turn pump on* [5].

As mentioned previously transitions connect steps with each other. Every transition has a guard, which determines by a boolean expression which valuation for the variables is allowed to take the transition. As soon as a guard

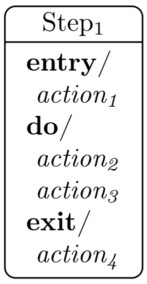


Figure 2.2: A schematic SFC step

is satisfied, the transition must be taken and the current step is deactivated. So transitions are urgent. If there are more than one outgoing transition with fulfilled guards, the one with the highest priority is taken. The priority of a transition is given by a partial order $\prec$.

The source and the target of a transition must not be a single step, but they can also be a set of steps. When having set of steps as source, all of these steps must be activated and the guard of the transition must be fulfilled before the transition can be taken. Thus, we have a synchronized deactivation of the source steps. If there is a set of steps as target of a transition the synchronization works just vice versa. The target steps are activated in parallel if the transition is taken. These two cases can appear separately as well as combined.

The previous description of an SFC retrieved from [2] and [5] leads to the following formal definition, which is also based on [2] and [5].

Definition 2.2.1 (SFC)**.** *A sequential function chart (SFC) is a 9-tuple $\mathcal{S} = (\mathit{Var}, \mathit{Steps}, \mathit{Actions}, s_0, \mathit{Trans}, \mathit{Blocks}, \sqsubset, \prec, \mathit{Hist})$, where*

- $\mathit{Var} = \mathit{Var}_I \cup \mathit{Var}_O \cup \mathit{Var}_L$ *is a finite set of variables,*
- *Steps is a finite set of steps,*
- *Actions is a finite set of actions, which can a variable assignment or an other SFC,*
- $s_0 \in \mathit{Steps}$ *is the initial step,*
- $\mathit{Trans} \subseteq (2^{\mathit{Steps}} \backslash \{\varnothing\}) \times G \times (2^{\mathit{Steps}} \backslash \{\varnothing\})$ *is a finite set of transition,*
- $\mathit{Blocks} : \mathit{Steps} \to 2^B$ *is a function which assigns a set of action blocks to each step,*
- $\sqsubset \subseteq \mathit{Action} \times \mathit{Action}$ *is a total order of the actions, which defines in which order the active actions must be executed,*
- $\prec \subseteq \mathit{Trans} \times \mathit{Trans}$ *is a partial order on the transitions,*
- $\mathit{Hist} \in \{0, 1\}$ *is a history flag, which determines whether a SFC is executed with history ($\mathit{Hist} = 1$) or not ($\mathit{Hist} = 0$).*

Semantics of an SFC Now we describe, how an SFC is executed on a *programmable logic controller (PLC)*. A PLC works in cycles. The duration of each cycle, the cycle time, varies between a lower bound δ_l and an upper bound δ_u. For the execution of an SFC the PLC performs the following steps in every cycle. Firstly the PLC gets the data from the environment and updates the corresponding values of the input variables. With the new valuation of the variables the PLC checks for every active step, if there are outgoing transitions, whose guards are satisfied by the valuation. From the set of the enabled transitions, the one with highest priority is taken. Now the PLC determines the set of actions, which are executed subsequently. The set consists of the actions with a *do* qualifier of all active steps and for every transition, that has been taken, the actions with an *exit* qualifier of the source and the actions with an *entry* qualifier of the target step. After executing these actions, the PLC sends the value of the output variables to the environment [5].

A more deeply description and the formal definition for the semantics can be found in [5].

Example 2.2.1. *Figure 2.3 depicts an SFC for pump P_1 of the plant example.*

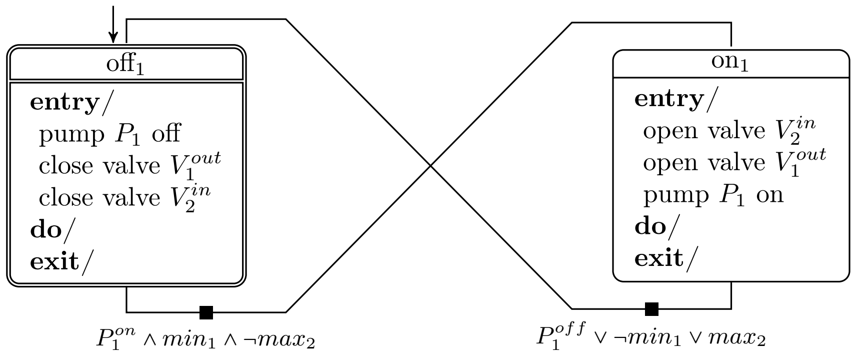


Figure 2.3: SFC for pump 1 from [2]

The initial step is off_1. When entering this step, three entry actions are performed. The first one is "pump P_1 off", which turns pump P_1 off. Afterwards the valve for the outgoing water flow is closed by the action "close valve V_1^{out}". Last the action "close valve V_2^{in}" closes the valve for the incoming water flow of the second tank. Now no water can leave tank T_1 and flow into tank T_2. Since there are no do or exit actions, the PLC is done, when it has executed the entry actions.

The PLC must leave this step, when the guard of the outgoing transition is fulfilled. The guard determines, that the step must be left, when the command to switch the pump on (P_1^{on}) is retrieved from the environment. Besides this condition tank T_1 may not be empty (min_1) and tank T_2 may not be full ($\neg max_2$). These informations are sent by the water level sensors of both tanks and assure, that neither pump P_1 can run dry nor tank T_2 can be flooded.

As soon as the guard of the transition is satisfied, the step on_1 is entered. Here the PLC opens firstly valve V_2^{in}, so that water can flow into tank T_2. Then the valve V_1^{out} is opened, so water can flow out of tank T_1. In the end the pump

P_1 is turned on. Now the water is piped from tank T_1 into tank T_2. Nothing more happens in this step, because there are no do or exit actions, like in the previous step.

For turning the pump P_1 off, thus getting to step off_1 again, one of the following conditions must be satisfied. The PLC can get the command to turn the pump off (P_1^{off}) or one of the water level sensors of the tanks indicates a problem. In this case either tank T_1 is empty ($\neg min_1$) and the pump will run dry or tank T_2 is full and will be flooded, if the pump pipes more water into the tank.

2.3 Hybrid sequential function charts

Ordinary SFCs can handle only discrete behavior, because the values for the variables can be set just by assignments of actions. These assignments are static and thus do not allow a continuous evolving of the values.

Therefore we extend an SFCs with a conditional *ordinary differential equation (ODE)* system to a *hybrid sequential function charts (HSFC)* with continuous behavior.

Firstly we describe the conditional ODE system.

2.3.1 Conditional ODE system

The components of a conditional ODE system are on the one hand a condition and on the other hand a set of ODEs. The condition specifies which values some variables from the SFC must have, so that an ODE is used to calculate a new value for a continuous variable by a function $f : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}$. This function computes the new value depending on the time t. For instance $\{f : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R} | \exists c \in \mathbb{R}. \forall t \in \mathbb{R}_{\geq 0}. f(t)(x) = 2t + c\}$ says the value of a continuous variable x is $x = 2 \cdot 4 + c = 8 + c$ for any $c \in \mathbb{R}$, if the time t is four time units. A short form to denote the function $f(t) = 2t + c$ for any $c \in \mathbb{R}$ is $\dot{x} = 2$.

The following definition is from [2].

Definition 2.3.1 (Conditional ODE system)**.** *Let Var_C be a set of continuous variables, ODE_{Var_C} the set of all ordinary differential equations over Var_C and Conds the set of all conditions.*

Then a conditional ODE system is a pair $(cond : ODEs)$, where $cond \in Conds$ and $ODEs \subseteq ODE_{Var_C}$ and $CODE_{Var_C}$ is set of all conditional ODE systems.

Example 2.3.1. *Assume we have the following conditional ODE system:*

$$ODE = (P_1^{on} \wedge \neg P_2^{on} : \dot{h}_1 = -c_1, \quad \dot{h}_2 = c_1)$$

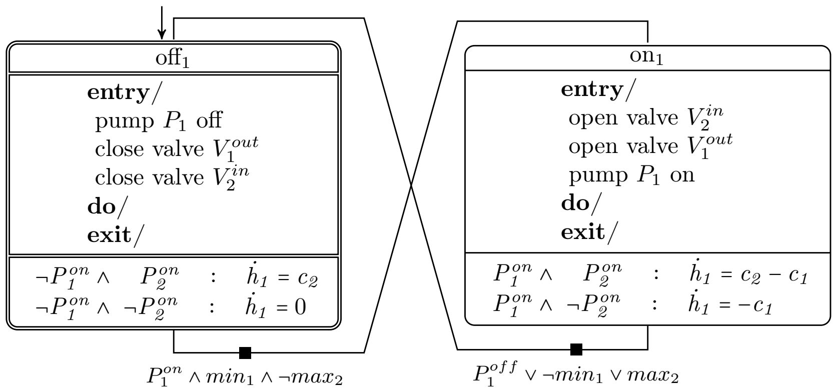
The condition of the system determines, that the command for turning on pump P_1 (P_1^{on})and the command for turning off pump P_2 (P_2^{off}) must have been received. If this condition is fulfilled, the water level of tank T_1 lowers according to the capacity c_1, which pump P_1 pipes out of the tank per time unit ($\dot{h}_1 = -c_1$). Corresponding to this the water level h_2 of tank T_2 increases by c_1 ($\dot{h}_2 = c_1$).

To extend an SFC to an HSFC, a set of conditional ODE systems is assigned to each step of the SFC. This extension is used to add continuous behavior to the chart.

The following definition is from [2].

Definition 2.3.2. *A hybrid SFC (HSFC) is a 10-tuple $HC = (Var, Steps, Actions, s_0, Trans, Blocks, Dyn, \sqsubset, \prec, Hist)$, where*

- *Steps, Actions, s_0, Trans, Blocks, $\sqsubset$, $\prec$, Hist are defined for SFCs,*
- *$Var = Var_I \cup Var_O \cup Var_L \cup Var_C$ is a finite set of variables,*
- *$Dyn : Steps \rightarrow CODE^{\star}_{Var_C}$ assigns a sequence of conditional ODE systems for continuous variables in Var_C to each step.*



off$_1$
entry/
pump P_1 off
close valve V_1^{out}
close valve V_2^{in}
do/
exit/
$\neg P_1^{on} \wedge P_2^{on}$: $\dot{h}_1 = c_2$
$\neg P_1^{on} \wedge \neg P_2^{on}$: $\dot{h}_1 = 0$
on$_1$
entry/
open valve V_2^{in}
open valve V_1^{out}
pump P_1 on
do/
exit/
$P_1^{on} \wedge P_2^{on}$: $\dot{h}_1 = c_2 - c_1$
$P_1^{on} \wedge \neg P_2^{on}$: $\dot{h}_1 = -c_1$
$P_1^{on} \wedge min_1 \wedge \neg max_2$
$P_1^{off} \vee \neg min_1 \vee max_2$

Figure 2.4: HSFC for pump 1 from [2]

Semantics of HSFCs A PLC executes an HSFC similar to the execution of an ordinary SFC. In every PLC cycle it performs the four steps updating the input variables according to the environment, determining the enabled transitions and taking them, executing the actions of the active steps and in the end sending the values of the output variables to the environment. Now we add a fifth step to the PLC cycle routine. After sending the values of the output variables to the environment, the PLC determines a set of ODEs, which must be executed. This set contains only these ODEs, which are attached an active step and whose condition is fulfilled by the values of the variables of the SFC. Afterwards the values of the continuous variables are computed and updated. Because these values evolve during the whole execution of a PLC cycle, the time t for the functions of the ODEs is exactly the cycle time. Thus, t is between δ_l and δ_u.

The previous description is based on [2]. A more deeply explanation and the formal definition can also be found in [2].

Example 2.3.2. *Figure 2.4 shows the SFC of tank T_1 from example 2.3, which is extended to an HSFC by ODE systems. The executing of this HSFC by a PLC is nearly the same as described in example 2.2.1 except the continuous behavior. By adding the ODE systems to the steps the changing of the water level in tank T_1 is calculated in each PLC cycle.*

Assume, the PLC is in step off$_1$ and has performed all actions of this step and has sent the output values to the environment. Now it calculates the new value for the water level h_1. Which ODE system is chosen for that is determined by the state of the pumps. Since we are in the step, where pump P_1 is turned off, both ODE systems have the condition $\neg P_1^{on}$. So the evolving of the water level depends on whether pump P_2 is turned on (P_2^{on}) or off ($\neg P_2^{on}$).

If pump P_2 is turned on, the condition $\neg P_1^{on} \wedge P_2^{on}$ of the first ODE system of step off$_1$ holds and the equation $\dot{h}_1 = c_2$ is responsible for the changing of the water level. This means, for the duration t of a cycle time the water level of tank T_1 increases by the amount of water pump P_2 pipes into the tank per time unit, which is given by c_2, the capacity of pump P_2.

The second ODE system of this step has the condition $\neg P_1^{on} \wedge \neg P_2^{on}$, so both pumps must be turned off. Now the water level h_1 does not change at all

($\dot{h}_1 = 0$), because in this case water is neither piped into nor out of the tank.

The second step on_1 has also two ODE systems. Since in this step pump P_1 is switched on, both systems have P_1^{on} as condition. Like in the first step, it depends on the state of pump P_2, which of them is chosen for the evolving of the water level.

If both pumps are turned on ($P_1^{on} \wedge P_2^{on}$), the water level changes according to the difference of the pump capacities c_1 and c_2 or short $\dot{h}_1 = c_2 - c_1$.

In the case, pump P_2 is turned off, the condition $P_1^{on} \wedge \neg P_2^{on}$ holds. Now water is piped out of the tank only ($\dot{h}_1 = -c_1$).

2.4 Hybrid Automata

A hybrid automaton is used to model and simulate systems with discrete and continuous behavior. In reality there are several examples for such a combination of both behaviors. Even the one tank system from the previous section 2.3 has a discrete part of behavior, for instance turning the pumps on or off, as well as a continuous part, the water level of the tank, which changes over time.

To keep track of the state of the system while simulating it a hybrid automaton has a finite set of variables Var and a finite set of locations Loc. The set Var is a union of two sets Var_{con} and Var_{dis}. The variables of the set Var_{con} refer to the continuous behavior. Usually they represent some sort of physical measurement, which evolves over time. In contrast to this, the variables of the set Var_{dis} can only have a specific number of values, because they belong to the discrete behavior of the system and thus the values do not evolve over time. Each variable has a value assigned to itself. To assign the values to the different variables in Var a function $v : \mathit{Var} \to \mathbb{R}$ called valuation is used. The set of all possible valuations for the variables in Var is V.

The set of locations also refer to the discrete behavior of the system, like the discrete variables, because a hybrid automaton may be in one single location at a time. It can switch between the different locations during the simulation, but it may never enter two locations at the same point of time. For this reason and because the set of locations is finite, the locations belong the discrete behavior.

As mentioned above the variables respectively the value assigned to them and the locations are used to determine the states of the system while simulating it. When the hybrid automaton is in a location loc and the valuation of all variables at this point of time is v, the state of the system is denoted by the pair (loc, v). The set of all states, a system can be in, is $\Sigma := \mathit{Loc} \times V$.

At the beginning of a simulation a hybrid automaton must have a set of states in which it can start. These states are the initial states of the system and they are defined by the set $\mathit{Init} \subseteq \mathit{Loc} \times V$.

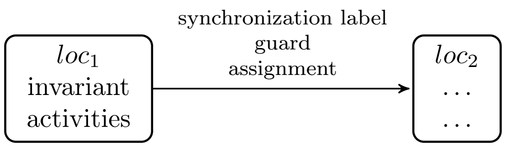


Figure 2.5: A schematic hybrid automaton

In figure 2.5 a schematic hybrid automaton is shown. The locations are depicted as rectangles with rounded corners. The figure shows, that an invariant and an activity is assigned to each location.

An activity of a location determines, how the values of the continuous variables change, while staying in the location and time is passing by. This is done by a function $f : \mathbb{R}_{\geq 0} \to V$, which assigns a new value to a continuous variable depending on the time. For instance $\mathit{Act}(loc) = \{f : \mathbb{R}_{\geq 0} \to V | \exists c \in \mathbb{R}. \forall t \in \mathbb{R}_{\geq 0}. f(t)(x) = 3t + c\}$ says the value of the variable $x \in \mathit{Var}_{con}$ is $x = 3 \cdot 4 + c = 12 + c$ for any $c \in \mathbb{R}$, if the controller stayed four time steps in the location loc. In the graphical representation of a hybrid automaton the activities are given by a differential equation for a better readability. So the set $\{f : \mathbb{R}_{\geq 0} \rightarrow V\}$ with $f(t) = 3t + c$ for any $c \in \mathbb{R}$ is specified by $\dot{x} = 3$.

The conditions, which define if the controller may enter a location respectively stay in it, are given by the invariants. An invariant of a location is a set of valuations for the variables, that are allowed in the location. For example $Inv(loc) = \{v \in V | v(x) \leq 3\}$ means that as long as the value of the variable $x \in Var$ is less or equal three, the valuation of the variables satisfies the invariant and the controller may stay in the location. But if for instance in the next time step the activities would change the valuations of the variables in a way, that the invariant gets violated, the controller must immediately leave this location. Thus, the controller must leave a location before the valuation can violate the invariant. For the graphical representation the invariants are given as first order formulas without quantifiers. For example, the invariant $Inv(loc) = \{v \in V | v(x) \leq 3\}$ would be denoted by $x \leq 3$.

As shown in figure 2.5 the locations of an hybrid automaton are connected via directed transitions or edges. These transitions have guards, assignments and a synchronization label assigned.

The guard of a transition is similar to the invariant of a location. It specifies a set of valuations the variables may have so the controller is allowed to take the transition. But in contrary to the invariants of locations, transitions are not urgent. So the controller can take the transition when its guard is fulfilled, but it is not forced to take it. Unless the invariant does not allow to stay one more time step in the current location and there are no other transitions whose guards are satisfied. In the graphical representation they are denoted like invariants, because they determine a set of allowed valuations.

There is also an equivalent to the activities of the locations, which are the assignments of an edge. But contrary to activities, the assignments just assign specific values to the variables in *Var*, which are not calculated by a function. The reason for that is, the controller cannot stay in an edge for an amount of time. It just uses it to get from one location to another.

Beside the guard and the assignment, an edge has a synchronization label. This label is important for the parallel composition and will be explained more precisely in section 2.4.1. A special edge is the τ-transition, which exists for every location of the hybrid automaton. For this transition the source and the target location are the same. Thus, it is a self loop. Besides the guard of this transition gives no restriction for the valuation of the variables. The assignment is given by *Id*, the identity set, which does not change any valuation of the variables. The τ-transition is as much important for the composition as the set of the controlled variables. This set is referred to as *Con* and contains all variables of a hybrid automaton, whose valuation are defined by an activity of a location in this automaton. Any other hybrid automaton is only allowed to read them. The advantage of such controlled variables becomes clearer in section 2.4.1.

The previous description based on [6] and [7] leads us to the following formal definition, which is retrieved from [7], [4] and [6].

Definition 2.4.1 (Hybrid automaton). *A hybrid automaton HA = (Loc, Var, Con, Act, Inv, Lab, Edge, Init) is a 8-tuple, where*

- *Loc is a finite set of locations,*

- $Var := Var_{con} \cup Var_{dis}$ *is a union of*

  Var_{con} *, a finite set of real valued, continuous variables,*

  Var_{dis} *, a finite set of discrete variables,*

- $Con := \{x \in Var | \exists l \in Loc, f \in Act, v, v' \in V, t \in \mathbb{R}_{\geq 0} : f(0)(x) = v(x) \wedge f(t)(x) = v'(x)\}$ *is a set of variables, whose evaluation is determined by an function of an activity assigned to a location in* $\mathcal{HA}$

- *Act is a function assigning a set of activities for the continuous variables* $f : \mathbb{R}_{\leq 0} \to V$ *to each location, which are time invariant, meaning that* $f \in Act(loc)$ *implies* $(f + t) \in Act(loc)$ *where* $(f + t)(t') = f(t + t')$ *for all* $t' \in \mathbb{R}_{\leq 0}$

- $Inv : Loc \to \mathcal{P}(V)$ *is a function assigning an invariant* $Inv(loc) \subseteq V$ *to each location* $loc \in Loc$*,*

- *Lab is a finite set of synchronization labels, including* $\tau \in Lab$*,*

- $Edge \subseteq Loc \times Lab \times Guard \times Ass \times Loc$ *is a finite set of edges, including the* τ*-transition* (loc, τ, V, Id, loc) *for each* $loc \in Loc$*,*

  - $Guard \subseteq V$ *is a set of valuations,*
  - Ass *is a set of functions* $g : Var \to \mathbb{R}$*,*

  where Id is the identity set $Id = \{v(x) \in V | v'(x) = v(x)\} \; \forall x \in Var$

- $Init \subseteq \Sigma$ *is a finite set of initial states,*

Semantics of a hybrid automaton At the beginning of an execution the hybrid automaton is in a state (loc_0, v_0) from the set of initial states *Init*. The valuation v_0 of the variables must satisfy the invariant of the initial location loc_0, so the hybrid automaton can enter the location. At this point and at any other point of the execution the hybrid automaton has two options of performing an action, either taking a *time step (flow)* or a *discrete step (jump)*.

Taking a time step means the hybrid automaton stays in the current location *loc*. While time is passing by, the valuation of the variables are changed by a function $f : \mathbb{R}_{\geq 0} \to V$. This function is determined by the activities $Act(loc)$, which are assigned to the location. Because no time has passed by when entering the location, the result of the function f with zero time units must be equal to the current valuation v. Thus, $f(0) = v$ must hold. After t time units the function f delivers a new valuation for the variables, which is $v' = f(t)$ and the system is in a new state (loc, v'). But the amount of time units in which the hybrid automaton is permitted to stay in the location, is restricted by the invariant $Inv(loc)$ which is assigned to the location. If and only if each valuation, which is delivered by the function f during the time interval $[0, t]$, is in the set of allowed valuations given by the invariant $Inv(loc)$, the hybrid automaton can stay in the location. From this it follows that, before the invariant gets violated, the hybrid automaton must leave the current location *loc*. It can leave the location earlier, but must leave it at latest, when the next time step would violate the invariant.

Leaving a location *loc* is done by performing a discrete step respectively a jump. Therefore at least one edge is needed, which leads from *loc* as the source location to any other location loc' as the target location. But the transition might be taken only if the following conditions holds. On the one hand the current valuation v must satisfy the guard of the transition. On the other hand after the assignments of the edge are performed the new valuation v' must be in the set of valuations, which are allowed by the invariant $Inv(loc')$ of the target location. If and only if these two conditions hold, the hybrid automaton can leave the current location loc and enter the new location loc'. The state of the system changes from (loc, v) to (loc', v'), when performing a discrete step.

The description of the behavior of the hybrid automaton based on [6] and [7] makes up the following semantics from [6]:

Definition 2.4.2 (Semantics of a hybrid automaton)**.** *The semantics of a hybrid automaton HA = (Loc, Var, Con, Act, Inv, Lab, Edge, Init) consists of two rules. One for the direct discrete steps from one location to another and one for the continuous time steps, while staying in a location.*

1. *Discrete step semantics*

$$\frac{e=(loc,a,g,ass,loc')\in Edge \quad v\in g \quad v'=ass \quad v'\in Inv(loc')}{(loc,v)\xrightarrow{a}(loc',v')} \; Rule_{discrete}$$

where $loc, loc' \in Loc, a \in Lab, g \in Guard$ *and* $ass \in Ass$

2. *Time step semantics*

$$\frac{loc\in Loc \quad f\in Act(loc) \quad f(0)=v \quad v'=f(t) \quad t\geq 0 \quad f([0,t])\in Inv(loc)}{(loc,v)\xrightarrow{t}(loc,v')} \; Rule_{time}$$

Example 2.4.1. *Assume we have the hybrid automaton HA_1 given by the figure 2.6. The automaton starts by setting the variable s to zero and than enters the first location loc_1. This location must be immediately left, because the invariant does only allow to stay in it, if $s \leq 0$. But since the activity $\dot{s} = 1$ increases the value of s for every time unit, it would violate the invariant, if the control stays in the location for one time unit.*

The guard of trans1 $s \leq 0$ of the outgoing transition is fulfilled, because s has still the value zero. So the second location loc_2 is entered. The invariant $0 \leq s \leq 60$ allows to stay in this location, as long as the value of s is between zero and 60. Since the activity increases the value of s by one per time unit, the control must not leave the location until 60 time units have passed by. The outgoing transition can only be taken, if s has exactly the value 60.

So we have to stay in loc_2 until 60 time units have passed, before we leave it. The value of s is then reset to zero and the control enters the initial state again.

The summary of the behavior of this automaton is, that it counts the variable s up to 60, resets it and starts again.

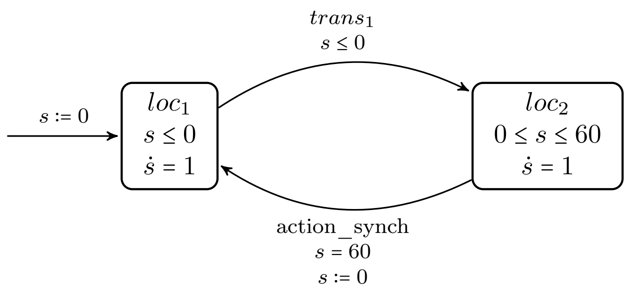


Figure 2.6: Hybrid automaton HA_1

2.4.1 Parallel composition

Large systems consist of several components respectively of several hybrid automata, that run in parallel. But for the analysis of such systems just a single automaton is required. Thus the parallel execution of the components of the system has to be modeled by a single automaton. This is done by the parallel composition.

To build the parallel composition $H_1||H_2$ of two hybrid automata H_1 and H_2 firstly the set of locations Loc is made up. Therefore each location from Loc_1 is combined with each location from Loc_2. For every new location $(loc_1, loc_2) \in Loc$ the invariants and activities are determined by building the intersection of the original invariants respectively of the activities of the locations $loc_1 \in Loc_1$ and $loc_2 \in Loc_2$.

Thus, the invariant of the new location $Inv((loc_1, loc_2))$ is a subset of the valuations, which are allowed by the invariant of both location loc_1 and location loc_2. If for instance the invariant of loc_1 is $\{x \leq 3\}$ and the invariant of loc_2 is $\{x \geq 2\}$, the resulting invariant is $Inv((loc_1, loc_2)) = Inv_1(loc_1) \cap Inv_2(loc_2) = \{v(x) \in V | 2 \leq v(x) \leq 3\}$ or simply $\{2 \leq x \leq 3\}$.

Nearly the same holds for the activities. Assume the activity of location loc_1 is $\{\dot{x} = 3\}$. From this activity we can get all valuations, where the new value of x is determined by the function $\dot{x} = 3$ and y can have a random value in $\mathbb{R}$. The same holds for the activity of loc_2, which is $\{\dot{y} = 2\}$. Here the value of x can be a random number in $\mathbb{R}$. Thus, in the intersection the value of both variables x and y are restricted by the given functions. The corresponding activity $Act((loc_1, loc_2)) = Act_1(loc_1) \cap Act_2(loc_2)$ in the composition is $\{(\dot{x} = 3) \wedge (\dot{y} = 2)\}$. A special case, that must be taken care of while building the composition, is if the two activities are contrary to each other. Assume, we have two activities $\{\dot{x} = 1\}$ and $\{\dot{x} = 2\}$ of two locations loc_1, loc_2. The intersection of these activities would be empty, which is not allowed and thus the location (loc_1, loc_2) is not allowed and would not be part of the set of locations in the composition.

After building the location set of the composition and its associated components, the transitions must be constructed. Here the synchronization labels play a decisive role. They are responsible for the interaction between the two hybrid automata H_1 and H_2. There exist two cases: Either the synchronization label in both label sets of H_1 and H_2, or it is just in one of them. If a synchronization label a is in the label set of both automata and one of the two automata takes an edge with this label, the other one must take it as well. So, the resulting edge $e \in Edge$ of the edge $(loc_1, a, g_1, ass_1, loc'_1) \in Edge_1$ of H_1 and the edge $(loc_2, a, g_2, ass_2, loc'_2) \in Edge_2$ of H_2 would be $e = ((loc_1, loc_2), a, g, ass, (loc'_1, loc'_2))$. The guard g of this new edge is the intersection of g_1, the guard of the edge of H_1, and g_2, the guard of the edge of H_2. So it is assured that an edge in $H_1 || H_2$ can only be taken, if the guard of the original edges of H_1 and H_2 are satisfied.

The assignment *ass* is built also via intersection. Let $\{x := 2\}$ be an assignment, where y has a random value and $\{y := 1\}$ an other assignment, where the value of x is random, then the intersection of these two would be $\{(x := 2), (y := 1)\}$. Similar to the intersection of the activities of an location, a transition in a composition is not allowed, if the intersection of non-empty assignments from the original edges is empty: for $\{x := 2, y := 2\}$ and $\{x := 1\}$ the intersection is empty. In this special case the transition is not allowed. Beside the intersection of the assignments, it must be paid attention to the variables in them. In the composition a variable x may only get a value $v'(x) \neq v(x)$ by an assignment, if x is in the set of the controlled variables of the original hybrid automaton, which contains the edge, whose assignment causes the change. The reason for this is that a variable may only be changed by an assignment of an automaton, which is allowed to change the valuation of this variable. This is specified be the sets of controlled variables of this automaton.

If a label is defined only for one automaton, in example $a \in Lab_1$ but $a \notin Lab_2$, H_1 takes a transition with the label a and H_2 takes a τ-transition. The corresponding transition in $H_1 || H_2$ is the following: $((loc_1, loc_2)$, a, g_1, ass_1, $(loc'_1, loc_2))$, because the guard of the τ -transition allows all valuation and the assignment does not change any variable. The same holds vice versa, if $a \in Lab_2$ and $a \notin Lab_1$.

Based on the previous description the formal definition of the parallel composition is as follows. Both are based on [6].

Definition 2.4.3 (Parallel composition)**.** *Let*

$H_1 = (Loc_1, Var, Con_1, Act_1, Inv_1, Lab_1, Edge_1, Init_1)$ *and*

$H_2 = (Loc_2, Var, Con_2, Act_2, Inv_2, Lab_2, Edge_2, Init_2)$

be two hybrid automata. Then the parallel composition $H_1 || H_2 = (Loc,\ Var,\ Con,\ Act,\ Inv,\ Lab,\ Edge,\ Init)$ *is a hybrid automaton with*

- $(loc_1, loc_2) \in Loc$, *iff*
  - $loc_1 \in Loc_1, loc_2 \in Loc_2$ *and*
  - $(Act_1(loc_1) = \varnothing \wedge Act_2(loc_2) = \varnothing) \Leftrightarrow Act_1(loc_1) \cap Act_2(loc_2) = \varnothing$
- $Con = Con_1 \cup Con_2$,
- $Act(loc_1, loc_2) = Act(loc_1) \cap Act(loc_2)$ *for all* $(loc_1, loc_2) \in Loc$,
- $Inv(loc_1, loc_2) = Inv(loc_1) \cap Inv(loc_2)$ *for all* $(loc_1, loc_2) \in Loc$,
- $Lab = Lab_1 \cup Lab_2$,
- $((loc_1, loc_2), a, g, ass, (loc'_1, loc'_2)) \in Edge$ *iff*

  - *there exists* $(loc_1, a_1, g_1, ass_1, loc'_1) \in Edge_1$ *and* $(loc_2, a_2, g_2, ass_2, loc'_2) \in Edge_2$*, such that*
  - *either* $a_1 = a_2 = a$ *or*
    $a_1 = a \in Lab_1 \setminus Lab_2$ *and* $a_2 = \tau$*, or*
    $a_1 = \tau$ *and* $a_2 = a \in Lab_2 \setminus Lab_1$ *and*
  - $g = g_1 \cap g_2$ *and*
  - $ass = ass_1 \cap ass_2$ *and*
  - $(ass_1 = \emptyset \wedge ass_2 = \emptyset) \Leftrightarrow ass_1 \cap ass_2 = \emptyset$ *holds.*

- $Init = \{((loc_1, loc_2), v) | (loc_1, v) \in Init_1 \wedge (loc_2, v) \in Init_2\}$.

Remark. *Locations, whose invariant, and transitions, whose guard, evaluate to false independent of the current valuation of the variables, can be omitted without changing the behavior of the parallel composition.*

Example 2.4.2. *Let HA_1 be the hybrid automaton from figure 2.6. A second hybrid automaton HA_2 is depicted in figure 2.7. Basically, it counts up a variable m till 60 by taking a self loop transition with the label action_synch. It resets the variable by an other transition, if m is exactly equal to 60.*

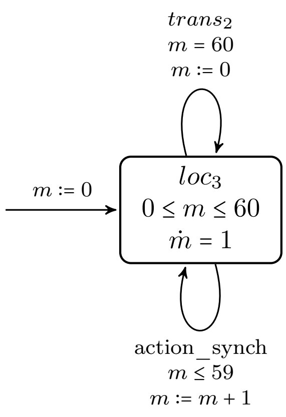


Figure 2.7: Hybrid automaton HA_2

The behavior of the parallel composition$HA_1 || HA_2$, which is depicted in figure 2.8, is as follows: The automaton still counts the variable s up till 60. When s is equal to this value, it is reset to zero and m is increased by one. If m reaches the value 60, it reseted to zero. So s counts the seconds of a minute and m the minutes of an hour.

To set up the parallel composition $HA_1 || HA_2$ we combine each location of HA_1 with the location of HA_2. Now we have two new locations (loc_1, loc_3) and (loc_2, loc_3) for the composition.

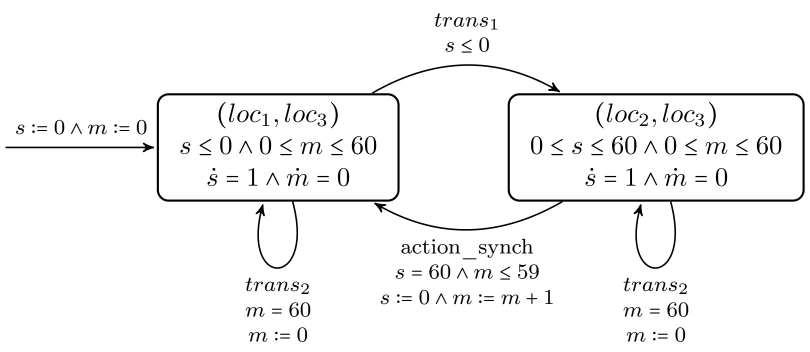

$trans_1$
$s \leq 0$
$s := 0 \wedge m := 0$
(loc_1, loc_3)
$s \leq 0 \wedge 0 \leq m \leq 60$
$\dot{s} = 1 \wedge \dot{m} = 0$
(loc_2, loc_3)
$0 \leq s \leq 60 \wedge 0 \leq m \leq 60$
$\dot{s} = 1 \wedge \dot{m} = 0$
action_synch
$s = 60 \wedge m \leq 59$
$s := 0 \wedge m := m + 1$
$trans_2$
$m = 60$
$m := 0$
$trans_2$
$m = 60$
$m := 0$

Figure 2.8: Parallel composition $HA_1 || HA_2$

For the invariant of (loc_1, loc_3), we build the intersection of the allowed sets from the original locations. Since $Inv(loc_1) = \{s \leq 0\}$ and $Inv(loc_3) = \{0 \leq m \leq 60\}$, we allow to stay in (loc_1, loc_3), if both invariants are fulfilled, thus $Inv(loc_1, loc_3) = \{s \leq 0 \wedge 0 \leq m \leq 60\}$. We do the same for the invariant of the location (loc_2, loc_3) and get $Inv(loc_2, loc_3) = \{0 \leq s \leq 60 \wedge 0 \leq m \leq 60\}$.

Because both locations of HA_1 have the same activity $\{\dot{s} = 1\}$ and HA_2's activity is $\{\dot{m} = 0\}$, we assigns the intersection $\{\dot{s} = 1 \wedge \dot{m} = 0\}$ of them as activity to both locations of the composition.

Now we build the transitions. The location loc_1 has one outgoing transition to loc_2 with the synchronization label $trans_1$. Since this label occurs only in HA_1, we add a transition from (loc_1, loc_3) to (loc_2, loc_3). We assign only the guard of the original transition to it, because there are no assignments for the original transition. The same holds for the transition in HA_2 with the label $trans_2$. We add it to both locations of the composition, because it is a self loop.

The label, which occurs in both automata, is action_synch. The corresponding transition in HA_1 leads from loc_2 to loc_1 and in HA_2 it is a self loop. So we add a transition to the composition from (loc_2, loc_3) to (loc_1, loc_3). The guard and the assignment are the intersection of the originals.

2.5 SFC verification tool

Since SFCs are used to control plants and wrong programed control systems can cause damages at a plant, the SFCs are safety critical. So it is important, that the control system works reliable.

To check if the SFCs for the controlling of a plant are safe, we can use the SFC verification tool. Additionally we need the behavior of the plant, which is given by conditional ODE systems. During the verification process our tool extends the SFCs with these systems to hybrid SFCs.

At this moment there exist no tool, which can perform a verification analysis on an HSFC directly. But we can transform an (H)SFC to a hybrid automaton, for which we have the opportunity of checking whether it is safe or not.

Therefore we use SpaceEx, a tool which analyzes hybrid automata. The result of the analysis could be, that the model is safe. If this happens, the verification is done and we do not have to take any further steps. But if the analysis reveals, that the model is unsafe, we have to do some CEGAR steps on the model. In this case SpaceEx gives us a set of bad states which should not be reachable. With this set and a heuristic, the SFC verification tool chooses a conditional ODE system and adds it to the SFC. Now the analysis is started again. These procedure is repeated until the system becomes safe or no ODE system is left to be added to the (H)SFC. In the latter case, the final result of the analysis is, that the system is unsafe.

A full description of the heuristic, which chooses the ODE system, can be found in [4].

In the following we explain the input files and their content, the transformation from an HSFC to a hybrid automata and some additionally needed components, which are added during the transformation more deeply. We give also a brief introduction of SpaceEx.

2.5.1 Input files

For the transformation of (H)SFCs to an hybrid automata, the SFC verification tool needs several informations about the system, which are given by the following input files:

plc.xml The *plc.xml* file contains a set of SFCs. They describe the behavior of the plant control, which should be analyzed by the SFC verification tool. We allow only SFCs without nested SFCs and without history flags.

The content of the file is based on the standard of programming interfaces for industrial automation by *PLCopen* [8]. This standard is product- and vendor-independent. To create the SFCs a program called *Beremiz* can be used. It is an integrated development environment for machine automation [9] with a graphical PLCopen editor.

connections.xml Here the connections between the in- and output variables of the SFCs are defined. For each variable, which must be retrieved from outside the SFC as input, it must be specified which SFC offers this variable as output.

control.xml Every plant can have the option for some user input, for instance switching the pumps on or off manually. The variables, that belong to a behavior, which is part of the normal behavior of the plant, are determined in this file.

condODE.xml The continuous behavior of the plant is specified by a set of conditional ODE systems, which is given in this file. Besides it is defined to which SFC of the system each ODE system belongs.

SpaceExConfig_ phav.cfg, SpaceExConfig_ supp.cgf For the analysis in SpaceEx some configuration options are needed. Besides a forbidden range for the values of a variable can be determined.

When starting the SFC verification tool all previous described files are read in and are transferred into different data structures. The SFCs are covered by a special data structure, which contains all the informations retrieved from *plc.xml*. Additionally the variables controlled by user input and the continuous variables are marked in this structure as well as the connection of the input variables to other SFCs. The ODE systems are covered by an own data structure. The same holds for the SpaceEx configuration options [4].

2.5.2 From HSFC to hybrid automata for SpaceEx

Since SpaceEx only analyzes hybrid automata, the SFC verification tool has to transform the HSFCs to hybrid automata. During the transformation, the SFC verification tool must take care of some restrictions given by SpaceEx. They will are explained later, when they become important in the transformation.

The transformation works as follows: The SFC verification tool creates an hybrid automaton for each SFC of the system by transforming

1. the variables
2. the locations
3. the transitions.

The following descriptions are retrieved from [2] and [4]. The formal definition of the transformation from an (H)SFC to a hybrid automaton can be found in [2].

Transformation of variables

To create the set of variables of an hybrid automaton, the SFC verification tool must be aware of different ways the IEC 61131-3 standard and SpaceEx define data types. The SFCs are created on the base of the IEC 61131-3 standard, which offers the following six variable types[1]:

VAR may only be changed by the SFC, it belongs to.

VAR_ INPUT can only be retrieved by an other SFC or by the environment. The variables may not be changed inside the SFC, which has it as input.

VAR_ OUTPUT sends its value to some SFC, where it is declared as input. The values of the variable can be changed inside its SFC.

VAR_ IN_ OUT is retrieved from some other component and may be changed inside its SFC. Besides it can be delivered to some other component of the system.

VAR_ GLOBAL is a global variable.

VAR_ EXTERNAL is retrieved from the configuration and may be changed inside the SFC.

However, SpaceEx only offers two flags to determine, which type a variables has. The *controlled* flag denotes a variable which may be changed by the SFC it belongs to and the *local* flag denotes variables, which can only be used internally in the SFC . The latter can neither be changed nor read by another component.

The SFC verification tool must transform the variable types of the SFC to the according types in SpaceEx. The variable type *VAR* becomes a *controlled* and *local* variable in SpaceEx, since *VAR* is only for inner use of the SFC and may only be changed by it. Variables of the type *VAR_INPUT* are retrieved from other components and hence not *local*. They are not *controlled*, because they may not be changed by the SFC. In contrary to this, variables of the type *VAR_ OUTPUT* are *controlled* because they may be changed by the SFC. But they are not local, because their values can be retrieved by other components. The same holds for *VAR_ IN_ OUT*, which is also transformed to *controlled* and not *local*.

Also the data types of variables are restricted by SpaceEx. It supports only integers and reals, while SFC variables can have other data types, for instance boolean. The SFC verification tool solves this by only supporting integers and reals and simulating booleans by integers, whose values may only be 1 for true and 0 for false.

Transformation of steps

The SFC verification tool creates the set of locations upon the set of steps of the SFC and the conditional ODE systems. For each step $s \in \mathit{Steps}$ it is checked whether it has some conditional ODE systems assigned or not.

In the simplest case, the step has no conditional ODE system and can be copied to the hybrid automaton as location. The resulting location has no invariant and no activity, because they can be retrieved from some ODE system only.

In the other case the step has some conditional ODE systems $\mathit{CODE} = (\mathit{cond_1} : \mathit{ODE_1}), \dots, (\mathit{cond}_n : \mathit{ODE}_n)$ assigned. Now $n+1$ copies $s_1, \cdots, s_{n+1}$ of the step s must be added to set of locations. Each copy s_i of $s_1, \dots, s_n$ gets an invariant, which is a conjunction of one condition $(\mathit{cond}_i : \mathit{ODE}_i)$ and the negation of the previously added conditions $\bigwedge_{j=1}^{i-1} \neg \mathit{cond}_j$. Building the invariant with the conjunction of the wanted condition and the negation of all previous conditions assures, that the location is only entered, when this one condition is satisfied. This is important, because the corresponding equation ODE_i is assigned to the copy s_i as activity and every ODE_i leads to another evaluation of the variables values.

The invariant of the last copy s_{n+1} is the conjunction of the negated conditions of all ODE systems $\bigwedge_{j=1}^{n} \neg \mathit{cond}_j$. This copy has no activity assigned and can only be entered, if no condition of any ODE system can be satisfied.

Figure 2.9 shows the schematic transformation of a step.

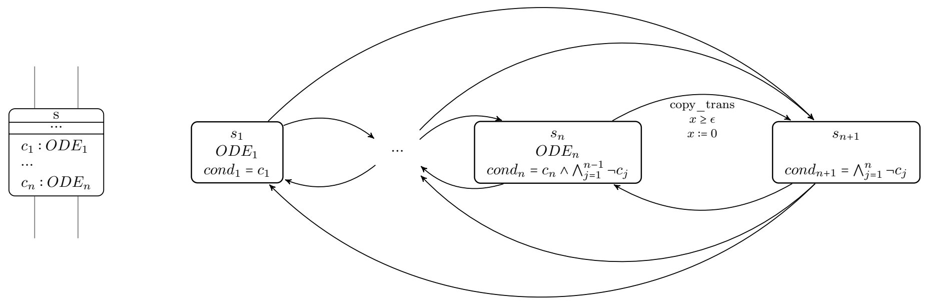


Figure 2.9: Transformation of an HSFC step

Because SpaceEx can handle only convex linear constrains for the invariants, the SFC verification tool must execute some modifications on the locations before they can be parsed to an input file for SpaceEx. Since no disjunctions and negations are allowed, the disjunctive normal form DNF_i is computed for the invariant $cond_i \bigwedge_{j=1}^{i-1} \neg cond_j$ of each location s_i. If there are negations of equations in the conjunctive clauses of the DNF_i, the SFC verification tool overapproximates them by replacing $\neg(x \geq y)$ with $(x \leq y)$, $\neg(x \leq y)$ with $(x \geq y)$ and $\neg(x = y)$ with (true). Afterwards the location s_i is replaced by a copy $s_{i,k}$ for every conjunctive clause $conj_k$ of the $DNF_i = conj_1 \vee \cdots \vee conj_m$ with $conj_k$ as invariant and ODE_i as activity. Figure 2.10 shows the locations after the modifications for SpaceEx.

In the end the SFC verification tool connects each copy $s_{i,k}$ with each other copy $s_{j,l}$, where $i \neq j$ or $k \neq l$, with a transition. So it is assured, that the control can switch between the different locations resulting from the step. This is needed, because in the original SFC it is possible to stay in the step s, but the valuation of the variables can change, so that another conditional ODE system is chosen to compute the values. Because the control can take these transitions infinitely many times within no time, we get Zeno behavior. To avoid this we add a new variable x. It must be greater or equal than some ϵ before a *copy_ trans* transition can be taken and is then reseted.

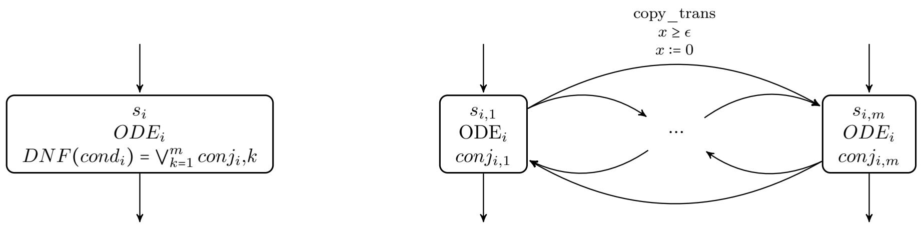


Figure 2.10: Applying SpaceEx restrictions

Transforming of transitions on locations

For every transition in an HSFC there is at least one corresponding transition in the hybrid automaton.

For every transition t_i in an HSFC, which connects a source and a target step, the SFC verification tool adds transitions $t_{i,1}, \ldots, t_{i,n}$ to connect every location, which has been made up by the transformation of the source step, with every location maintained by the transformation of the target step.

Figure 2.11 shows the transformation of the transitions. For convenience the source step is not split during the the transformation of the locations.

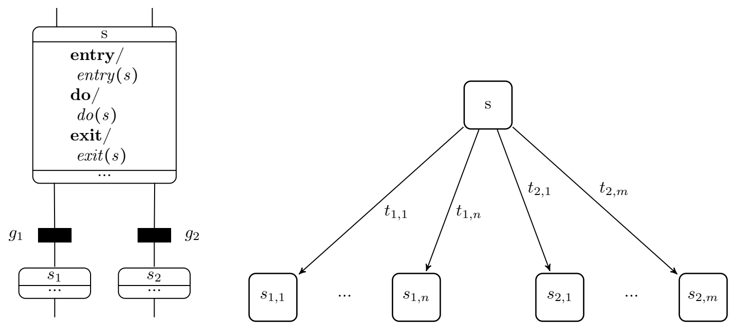


Figure 2.11: Transformation of transition without guard and assignment

Each transition $t_{i,k}$ of $t_{i,1}, \ldots, t_{i,n}$ gets some actions assigned to it as assignment, because both only set values to the variables and do not calculate them depending on time like the activities of an location does. To simulate the behavior of an SFC when switching from a source to a target step, the SFC verification tool assigns to a transition $t_{i,k}$ in the hybrid automaton the *exit* actions of the source step and the *entry* and *do* actions of the target step. In an SFC, the *do* actions of a step can be executed as long as the PLC stays in the step. Thus a self loop with these actions is added to every location achieved from the step transformation.

All these transitions have the synchronization label *action_ synch*, because when taking this transition a PLC cycle must be ended and the output variables must be sent to the environment. For updating the input variables at the beginning of a new PLC cycle, every location has a second self loop with *read_ synch* as synchronization label. This transition has no guard, because it must be allowed to take it at any time, when the control is forced to do so.

This does not hold for the *action_ synch* transitions. In an SFC the step can only be left, if the guard of the transition t_i is fulfilled. To achieve the same behavior in the hybrid automaton the guard must be transferred to the corresponding transitions $t_{i,1}, \ldots, t_{i,n}$. In addition to that the order $\prec$ of taking for transitions in the SFC must be respected. Therefore the transitions $t_{i,1}, \ldots, t_{i,n}$ corresponding to the outgoing transition t_i of a step with highest priority have only the guard of t_i. In an SFC any other outgoing transition t_j with $j \neq i$, which has a lower priority ($t_i \prec t_j$), can only be taken, when no transition with a higher priority can be chosen. To guarantee this behavior, the corresponding transitions $t_{j,1}, \ldots, t_{j,n}$ get additionally to their own guard, the negation of all guards from higher prioritized transitions.

Since in an SFC a transition must be taken as soon as it is enabled, we have to assure, that the hybrid automaton does the same. So it must be prevented, that the control always takes the self loop for the *do* actions, when an *action_ synch* is performed, instead of taking on outgoing transition. To achieve this, the guard of this self loop is the conjunction of all guards from outgoing transition negated. Figure 2.12 shows the schematic transformation of the transitions.

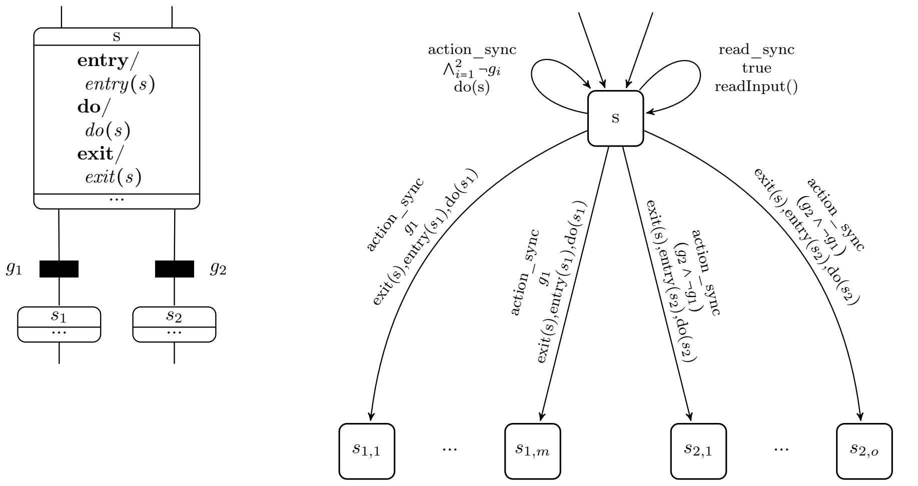
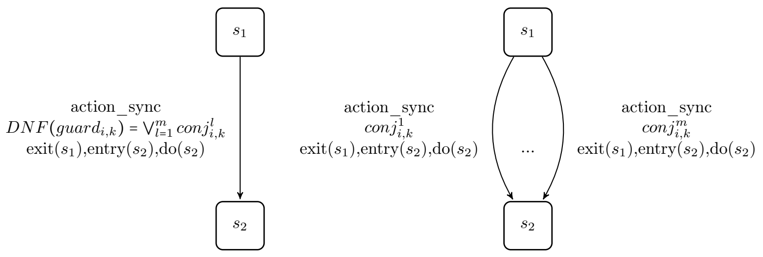


Figure 2.12: Transformation of transitions

Like for the invariants of the locations, also for the guards of the transitions SpaceEx allows no disjunctions and no negations. So the disjunctive normal form $DNF_{i,k}$ is computed for the guard of every transition $t_{i,k}$. The negation of an equation in every conjunctive clause $conj_{i,k}^{l}$ of $DNF_{i,k} = conj_{i,k}^{1} \vee \cdots \vee conj_{i,k}^{m}$ is overapproximated. Now the transition $t_{i,k}$ is replaced by m transitions with the same label and assignment, but with $conj_{i,k}^{l}$ as guard. Figure 2.13 shows the transitions after the modification for SpaceEx.



s_1
action_sync
$DNF(guard_{i,k}) = \vee_{l=1}^{m} conj_{i,k}^{l}$
exit(s_1),entry(s_2),do(s_2)
s_2
s_1
action_sync
$conj_{i,k}^{1}$
exit(s_1),entry(s_2),do(s_2)
...
action_sync
$conj_{i,k}^{m}$
exit(s_1),entry(s_2),do(s_2)
s_2

Figure 2.13: Applying SpaceEx restrictions on transitions

2.5.3 System components

During the transformation some additional hybrid automata are created. An automaton *timer* can optionally be added to measure the runtime of the system. To simulates the user input by randomly setting and resetting the user input variables a *controller* automaton is added as well as a synchronization automaton. The latter simulates a PLC cycle and synchronizes the automata of the system, when reading the input and writing the output. Additionally SpaceEx needs a network component. It specifies for each input variable of an automaton of the system to which output variable of which automaton it belongs.

For a better understanding we describe how the synchronization automaton simulates the PLC in the following.

Synchronization automaton

When a PLC executes several SFCs in parallel, retrieving the input variables from the environment is done synchronized for all component. The same holds for writing the output variables and sending them to the environment. To simulate this behavior for the hybrid automata an additional automaton is used. Figure 2.14 shows this synchronization automaton.

A PLC starts a cycle by getting the values for the input variables from the environment, so the automaton has *read_ synch* as initial state. The variable t measures the time of the PLC cycle and is set to $t := 0$ when starting the automaton. Because $t \leq 0$ is the invariant and $\dot{t} = 1$ the activity, the control must immediately leave the location *read_ synch*, because staying just one time unit in this location would violate the invariant. The only outgoing transition has the guard $t \leq 0$, but since the control has left the location *read_ synch* immediately, the value of the cycle time is $t = 0$ and the guard is satisfied. By taking the transition, which has the synchronization label *read_ synch*, all other hybrid automata of the system are forced to take their transition with this label and perform the synchronized updating of the input variables.

Afterwards the control enters the location *action_ synch*. Here the variable t increases accordingly to the time that passes by because of the activity $\dot{t} = 1$. The control may stay in this location as long as t does not violate the invariant, thus as long as t is below or equal to δ_u, which is the upper bound for the duration of a cycle. The location can be left earlier, since the guard of the only outgoing transition is fulfilled once t is between the lower bound δ_l and the upper bound δ_u of the duration of a cycle. Hence, the invariant of the location *action_ synch* and the guard of the outgoing transition assure that the cycle time t can vary between these two bounds, but can never be less than δ_l or more than δ_u.

The outgoing transition is not only important for the cycle time. It also simulates the last step of an PLC cycle, which is sending the values of the output variables to the environment. This can be happen in two ways. In the one case the cycle time reaches the upper bound and the synchronization automaton takes the transition with the synchronization label *action_ synch*. In this case all other automata of the system must perform the sending of the output values at the same time by taking their transitions with this label. In the other case an other automaton of the system takes its transition with the label *action_ synch* and thereby forces all other automata, including the synchronization automaton, to do the same.

Once this transition is taken by the synchronization automaton, the cycle time t is set to zero and a new cycle starts.

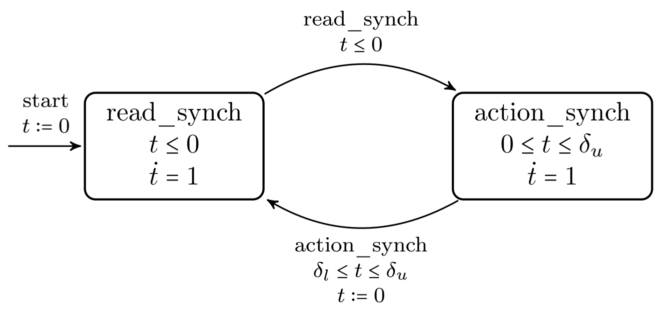


Figure 2.14: Hybrid automaton for synchronization of PLC cycle from [2]

2.5.4 SpaceEx

After the transformation of the (H)SFCs to hybrid automata and adding the additionally needed components, we give the resulting input file to SpaceEx. It performs a reachability analysis for the given system. During the analysis, it checks if the values of the variables can reach a forbidden range. The range is specified int the configuration file for SpaceEx.

For the reachability analysis SpaceEx can use the PHAVer method [10] or a scenario based on the representation of the state space by support functions [11]. For piecewise constant dynamics the PHAVer method achieves good results and for affine dynamics the support functions are more suitable [4].

SpaceEx is able to handle several hybrid automaton which runs in parallel. Therefore it computes the composition on the fly during the analysis. It takes the initial location and build all locations of the composition, that can be reached from it via transitions. After adding them, it checks if they can be reached. Only locations which are reachable considered in the further analysis.

To prevent chaotic evaluation of the variables during the analysis, SpaceEx assigns for each variable an activity to each location which determines that the value of a variable does not change unless there exists already an activity for this variable. The same holds for the assignments. If there is no assignments for a variable, SpaceEx adds an assignment which marks that the variable keeps its value when the corresponding transition is taken.

A more deeply description of SpaceEx can be found in [3].

Chapter 3

Data structure for hybrid automata

The SFC verification tool supports SpaceEx as the only analysis tool. During the development of our tool, we decided to make it more flexible, so that we are able to use other analysis tools for our verification.

The SFC verification tool saves only the SFC and the ODE systems in a special data structure. The automata needed for the analysis are created and the restrictions of SpaceEx are applied during this creation. The resulting automata are directly written in an input file for SpaceEx. Thus, they are not saved in a data structure. So it is not possible to transform the automata for others tools, which need individual input file with other restrictions to the automata than SpaceEx.

To realize the flexibility we introduce a new data structure for hybrid automata, which allows us to assign some changes according to chosen the tool for the analysis afterwards.

Therefore we implemented some *Creator* classes which perform the transformation from the (H)SFCs to hybrid automata. We omit here the description of the transformation, because it was already explained in section 2.5.2. The difference is, that the transformation omits the restrictions given by SpaceEx when building the automata and stores them in the data structure instead of directly writing them in the input file. Thus, we obtain general automata that are independent of any tool syntax. Moreover, we implemented a tool for our toolchain, that provides the SpaceEx specific syntax and applies the necessary changes to the automata data structure. To create an input file for an analysis tool, we just have to apply the necessary tool to the toolchain and implement a translation from the automata data structure into the syntax of the corresponding tool.

For saving the hybrid automata, we introduced the following data structure which is depicted in figure 3.1. The getter and setter methods are omitted for a better readability.

Class *Automaton* Each hybrid automaton is represented by an object of the class *Automaton*. It uses several classes to save the components, which belong to the automaton. Every automaton has an unique *id*. This *id* is either the name of the SFC, from which it is built, or the name of the additional components, which

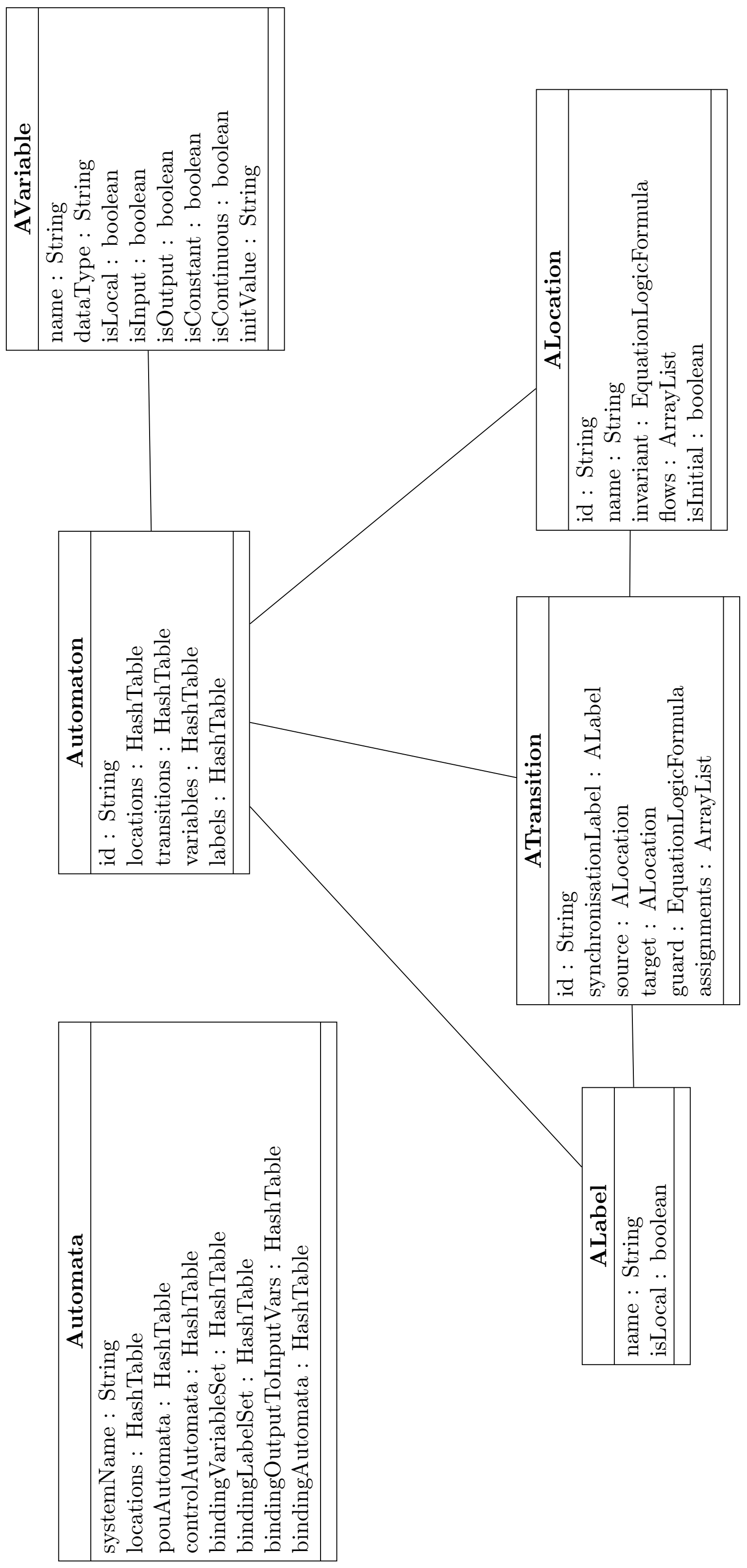


Figure 3.1: Class diagram of the automata data structure

are needed for the simulation of an PLC run, for instance the *synchronizer*.

Class *AVariable* The set of variables is stored in a table, where every variable is an object of the class *AVariable*. An object of this class has a name and a data type. Beside this some booleans determine, if the variable is only local or used as input or output variable. It is also specified, if it is continuous or constant, so can never change its value after the initialization. The initial value of the variable is saved in this class too.

Class *ALabel* All labels, which are used in the automaton, are saved in a table as object of the class *ALabel*. This class determines the name of the label and whether it is used locally or globally as a synchronization label.

Class *ALocation* A location of the automaton is an object of the class *ALocation*, which has an *id* and a *name*. The name is a combination of the step name and the id for a better readability, when looking at the results, while the id is only for the inner use in the program. Additionally, the invariants and the activities, here called flows, are saved in this class. A boolean determines, if it is an initial location or not.

Class *ATransition* The transitions, which connects the locations with each other, are also stored in a table. They are objects of the class *ATransition*. It has an *id* for differing the transitions from each other. The source and the target are saved as objects of the class *ALocation*. Besides the guard and the assignments belonging to the transition are save here.

Class *Automata* Since we retrieve several automata from the SFCs, there is a class *Automata*. It saves all automata, which belong to the system, separated in those, who are based on SFCs, and those, who are needed to simulate the PLC or the user input. Beside the automata, we have three tables to store informations for connection between them. For every input variable it must be determined to which output variable it belongs. The first tables contains all automaton which have in- or output variables. All in- and output variables of the system are saved in a second table. And which input variable belongs to which output variables is determined in the last table.

The advantage of this data structure is, that it is independent from SpaceEx. So we can make up a new package of classes, which apply easily the restriction and specifics any analysis tool needs later.

Chapter 4

Building the composition

Since the goal is to make the SFC verification tool more flexible for the integration of other analysis tools, we have to take care of the fact that there are analysis tools which do not build the parallel composition on their own. For this reason, we introduced a class *CompositionCreator* for it. While working on the project, we discovered, that SpaceEx has some problems with the composition and does not always compute it correct. A comparison of the compositions built by SpaceEx and by our tool can be found in chapter 5.

The class *CompositionCreator* gets an object of the class *automata*. From this it makes up a list with all automata of the system. In the first run it takes two automata from the list and starts computing the parallel composition of it. Afterwards the currently created compositions and a next automaton from the list are taken and the composition of these two is built. This is repeated until no more automata are left in the list.

While building the parallel composition, we start with the locations. Here we combine each location of the first automaton HA_1 with each location of the second automaton HA_2.

For the new invariant, we check first, if one of the original invariants is empty. If this is the case, we do not need to compute an intersection and just take the non-empty invariant for the new location. Even more simpler is the case for two empty invariants. Here the new invariant is also empty. But if both are not empty, we build the intersection of these two. Therefore we combine the two invariants with a conjunction. To minimize the new invariant, we compute the disjunctive normal form of it. Now we take every conjunctive term of this formula and remove double occurring equations. To complete the minimization of the formula, we search for every conjunctive term t_1, which contains another term t_2 and erase t_1. We can do this, because the set of valuations which fulfill t_1 is a subset of the set of valuations which fulfill t_2. If we have for instance the DNF $(a \wedge b) \vee (a \wedge b \wedge c)$, it is enough to keep $(a \wedge b)$. It may happen, that we get *false* as result of the intersection of the two invariants. In this case, we do not add the location to the composition, because it can never be entered.

The activity for the location is also computed by the intersection of the two original activities. Therefore we check, if one of them is empty and add the non-empty one to the location. If both activities are not empty, we compare each single equation from one activity with each equation from the other. If they are double we add just one and if they differ we add both equations. If we find two equations, which try to change the same variable in different way, we set a flag, that symbolizes, that the intersection is empty. If this flag is set, we know that we have an empty intersection from two non-empty activities and do not add the location to the composition according to definition 2.4.3.

After we have built the locations, we make up the variables of the composition. Therefore we check, if a variable occurs in both automata. If it is just in one of them, we add it to the composition. A variable, which is part of both automata, must eventually be changed in its relation to the whole system, before it is added to the composition. Only if it is assigned to be local in both automata, it is a local variable in the composition, too. This holds not, if the variable is an in- or output variable in at least one automaton. In this case it must be possible to read or set the value of the variable by another automaton, which would not be allowed, if we assign it as a local variable. For the case, that it is an input respectively an output variable in one of the two automata, the same holds for the composition.We assume, that the data type, the initial value and the continuous and constant flag, are the same in both original variables.

The labels of the composition must also contain all labels from both automata. A label, which is just in the list of one automaton is directly added to the composition. For the labels, which occur in both automata, we have to distinguish, if the original label are both local. Only then they are added as local labels to composition. Otherwise they are synchronization labels.

Now we have to build the transitions. For every location, we added to the composition we take the two original locations and retrieve all outgoing transitions from them. Each transition of the one location is compared to each transition of the other location. If they have the same synchronization labels, we build the intersection of the guards respectively the assignments like we did it for the location. We add them to composition only, if the new guard is not *false*, thus the transition can be taken, and if the intersection of the assignments is not empty because of contrary assignments to some variables. The new transition leads from the location for the two original locations to the location of the two targets of the original transitions. If the synchronization label is only part of one automaton, we take the transition and build a new one, which leads from the location for the two original locations to the location, which is the combination of the target of the original transition and the other location. For the guard and the assignment we take the original ones of the transition.

In the end we perform a depth-first search starting from the initial location of the composition. All locations which can not be reached via transitions are detected and deleted to keep the composition as small as possible.

Example 4.0.1. *Assume we have the synchronization automaton which is shown in figure 2.14. Additionally we have an automaton control_panel depicted in figure 4.1. It sets randomly the variables $P1\_off\_request$ and $P1\_on\_request$ by taking the var_set transitions. These variables represent the user input and trigger switching a pump P_1 on or off. They are reseted by taking the read_synch transition. A more deeply description of the function of this automation can be found in section 5.2.1.*

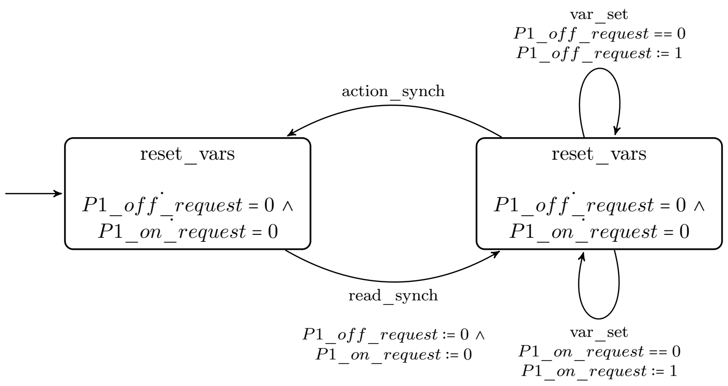


Figure 4.1: Hybrid automaton for simulating the user input for pump P_1

We use the CompositionCreator to compute the parallel composition (synchronizer || control_panel). We start with making up the locations. Therefore we combine each location of the synchronization automaton with each location of the control_panel and compute the intersection of their invariants and the intersection of their activities. First we take the read_synch location from the synchronization automaton and the reset_vars location from the control_panel. Since the latter has no invariant, thus all valuation are allowed in this location, we can just assign the invariant of the read_synch location to the new location. For the intersection of the activities we combine the two original activities by a conjunction, because we have no contrary or double assignments. This holds for all locations of the composition, because the activities in both locations of the synchronization automaton are equal as well as the activities of the locations in the control_panel. Since both original locations are the initial locations of their automata, the corresponding location in the composition is also the initial location. The location which is the combination of the read_synch location and the control location has the invariant $t \leq 0$*, since the control location has no invariant. Similar to this, the locations (action_synch, reset_vars) and (action_synch, control) have both the invariant* $0 \leq t \leq delta\_u$*, because the locations reset_vars and control have no invariants. The resulting locations are shown in figure 4.2.*

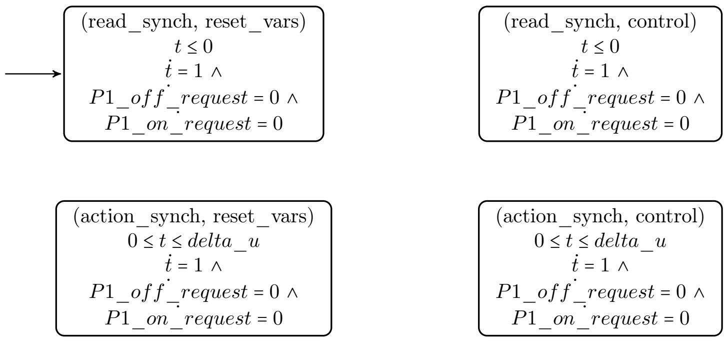


Figure 4.2: Locations of the composition (synchronizer || control_panel)

The variables of both automata can simply be assigned to the composition, since no variable occurs in both automata. For the labels we have the action_synch label and the read_synch label which are both synchronization labels, because they were not local in the original automata. This hold not for the var_set label. It is assigned to to the composition as a local label as in the control_panel, since it is no label in the other automaton.

Now we create the transitions of the composition. For the location (read_synch, reset_vars) we look up the outgoing transitions from the original locations. The read_synch location has a transition with the synchronization label read_synch leading to the action_synch location. Also the reset_vars location has a transition with the same label. The target of this transition is the control location. Because both transitions have the same label, we add a transition to composition which leads from (read_synch, reset_vars) to (action_synch, control). Since only one of the original transitions has a guard, we assigns it to the new transition. The same holds for the assignments. The transition leading from the location (action_synch, control) to the location (read_synch, reset_vars) is built similar. Since the action_synch location has two self loops both with the label var_set which is not part of the synchronization automaton, we add these self loops to the (action_synch, control) location and the (action_synch, reset_vars) location. For the (read_synch, control) location no transitions are added, because the read_synch location has an outgoing transition with the synchronization label read_synch which is a part of both automata, but the control location has no such outgoing transition. The same holds for the outgoing transition of the control location with the synchronization label action_synch. For the same reason the (action_synch, reset_vars) location has no outgoing transition leading to other locations. The resulting automaton is depicted in figure 4.3.

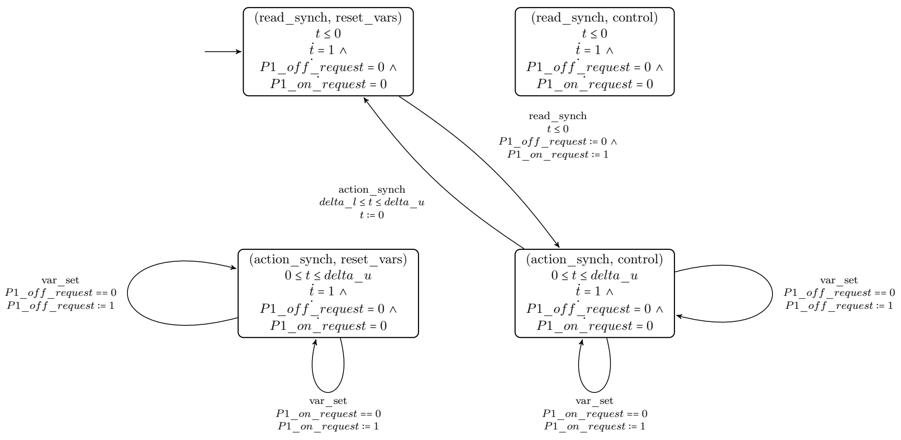


Figure 4.3: Composition (synchronizer || control_panel)

Now we look for the states that are not reachable. The locations (read_synch, reset_vars) is the initial location, so we do not have to check it for reachability. The location (action_synch, control) is reachable, because it has an incoming transition from the initial locations. The locations (read_synch, control) and *(action_ synch, reset_ vars) are removed from the composition, since they have either no incoming transition or only self loops. The final composition is shown in figure 4.4.*

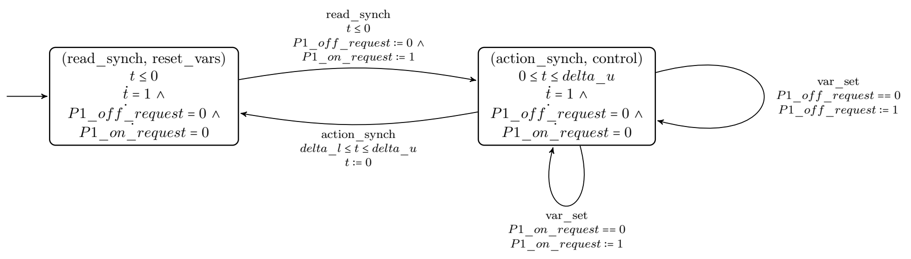


Figure 4.4: Final composition (synchronizer || control_panel)

Chapter 5

Testing the parallel composition

In this chapter we introduce two benchmarks and do a comparison between our implementation of the parallel composition and the parallel composition SpaceEx computes on the fly during the analysis.

5.1 Benchmarks

5.1.1 System with one tank

The first benchmark we use is very simple. The system consists of one tank which is filled with water and a pump *P_1*. The pump can be switch on or off by a user. If it is switched off, the water runs continuously out of the tank. But if the pump is switched on, it replaces the water that runs out so the water level is kept always at the same height.

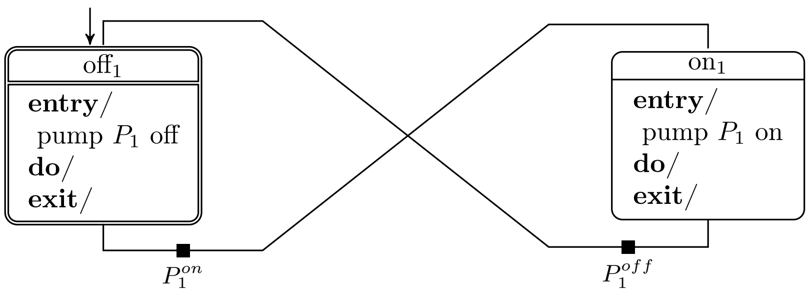


Figure 5.1: SFC for pump

The SFC of the system is shown in figure 5.1. There are two steps, one for the case the pump is switched off (*off$_1$*) and one to show, that the pump is working (*on$_1$*). We assume, that the pump is turned off at the beginning, so the step *off$_1$* is the initial step. Both steps are connected with each other by two transitions. The first one leads from step *off$_1$* to step *on$_1$*. The transition must be taken, if the user gives the command to switch the pump on. The command is represented by the variable P_1^{on}, which is assigned to be *true*, if the pump should be turned on. Vice versa the variable P_1^{off} corresponds to getting the command from the user to switch the pump off. If this command is received, thus P_1^{off} is *true*, the transition which leads from step on_1 to step off_1 must be taken. In the steps we have each one *entry* action, *pump P_1 off* in step off_1 and *pump P_1 on* in step on_1. They set a variable *chbk_P1_on* which gives a check-back if the pump is currently running or not.

For the continuous behavior of the system, we have the following set of conditional ODE systems:

$$(chkb\_P1\_on \quad : \quad \dot{h_1} = 0)$$

$$(\neg chkb\_P1\_on \quad : \quad \dot{h_1} = -1)$$

They determine the change of the water level h_1 in the tank during the execution of the system depending on the state of the pump. If the pump is not running, height of the water level decreases by one per time unit ($\dot{h}1 = -1$), since now water is running out the tank without being replaced by new incoming water. This changes, if the pump is turned on. Now the water level does not change ($\dot{h}1 = 0$), because the incoming water piped in by the pump refills what runs out.

Beside the SFC and the conditional ODE systems, we give to the SFC verification tool a file that marks the variables P_1^{on} and P_1^{off} as control variables, because they represent some user input for the system.

5.1.2 System with two tanks

This benchmark is very similar to the plant example in figure 2.1, except that we do not model the valves and the sensors for the water level. The remaining system consists of two tanks T_1 and T_2 which are filled with water and are connected to each other by pipes. We have two pumps P_1 and P_2. The pump P_1 pipes water from tank T_1 into T_2. Vice versa the pump P_2 pipes water from tank T_1 back into T_2. The pumps can be switched on or off by a user.

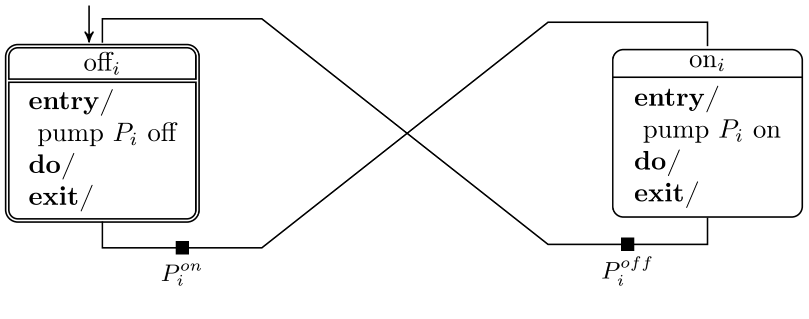


Figure 5.2: SFC for pump P_i with $i \in \{1,2\}$

The SFC for the pump P_1 and P_2 are shown in figure 5.2. They behave like the pump from the previous section. So we have two steps, one for each state of a pump P_i, (off_i) and (on_i). At the beginning, both pump are turned off, so the step off_i is the initial step in both SFCs. The two transitions of the SFCs have the same effect as in the system with one tank. We can switch the locations with them, if a running pump P_i is turned off or an idle pump P_i is turned on.

For the continuous behavior of the system, we have the following set of conditional ODE systems:

$$(\neg chkb\_P1\_on \wedge \neg chkb\_P2\_on \quad : \quad \dot{h}_1 = 0 \wedge \dot{h}_2 = 0)$$

$$(\neg chkb\_P1\_on \wedge chkb\_P2\_on \quad : \quad \dot{h}_1 = 2 \wedge \dot{h}_2 = -2)$$

$$(chkb\_P1\_on \wedge \neg chkb\_P2\_on \quad : \quad \dot{h}_1 = -1 \wedge \dot{h}_2 = 1)$$

$$(chkb\_P1\_on \wedge chkb\_P2\_on \quad : \quad \dot{h}_1 = 1 \wedge \dot{h}_2 = -1)$$

They determine the change of the water levels h_1 and h_2 for both tanks during the execution of the system. If no pump is running, the height of the water level does not change. If only pump P_2 is switched on, the water level decreases by 2 in tank T_2 per time unit and the water level of tank T_1 raises by 2. Vice versa the height of the water in tank T_1 lowers and the one of tank T_2 raises by 1 per time unit, if only pump P_1 is running. Since pump P_2 has a higher capacity than pump P_1, the water level of tank T_1 increases, while the water in tank T_2 becomes less, if both pump are switched on.

Beside the SFC and the conditional ODE systems, which we give to the SFC verification tool, we mark the variables P_1^{on}, P_1^{off}, P_2^{on} and P_2^{off} as control variables, because they represent some user input for the system.

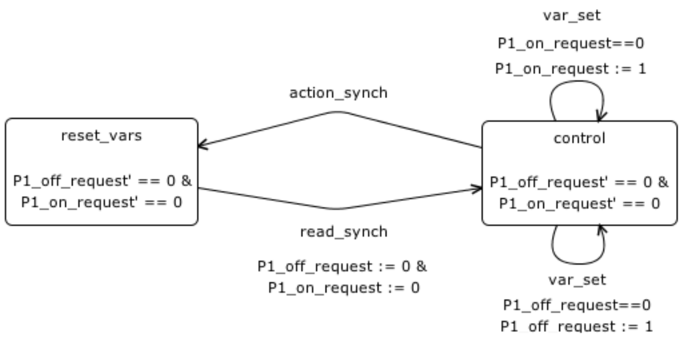


Figure 5.3: Hybrid automaton *control_panel*

5.2 Experimental results

5.2.1 Parallel composition for system with one tank

When we start the SFC verification tool, it reads in all the informations described in section 5.1.1. Upon the given informations it builds a set of three automata. One is the synchronization automaton shown in figure 2.5.3, which simulates the PLC cycle. We omit the description, because its behavior and function was already explained in section 2.5.3.

The second automaton is the *control_panel*, which is shown in figure 5.3. It simulates the random user input. This is done by randomly taking one of the *var_set* transitions, which are self loops of the location *control*. These transitions set the control variables, that represent a request for either switching the pump on or off. Via an *action_synch* transition we reach the other location of this automaton. The transition has no effect in contrary to the *read_synch* transition, which leads back to the *control* location. Here the control variables for sending command to pump are reseted. So we are able to set them randomly again, when we reenter the *control* location. Both locations have only activities, which determine that the variables of the automaton can not be changed, when we a take time step in the location. This assures, that they are only changed by the *var_set* transitions and keep their values until they have been transmitted to the pump by the *read_synch* transition.

The last automaton of the set is *tank1*, which simulates the behavior of the tank and the pump. Figure 5.4 shows the resulting hybrid automaton, which was build from the given input files.

The invariants and the activitie of the locations are the same. The invariants denotes, the water level h_1 may vary between -100 and 100. The activities assure, that neither the incoming command to switch the pump on or off is changed by the activities nor the state of the pump itself as long as we are in a location. So the pump must keep running or stay turned off until we get another command when taking the *read_synch* transition.

This transition is a self loop at both locations. It reads the input values and assigns the new values to the variables *P1_on* and *P1_off*. These two variables trigger the execution of the *action_synch* transitions. We have one of them at each location as self loop. They are taken if the pump does not change

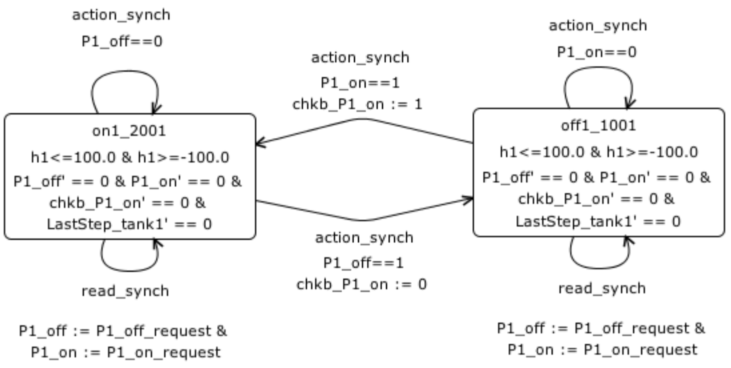


Figure 5.4: Hybrid automaton *tank1*

its state. For instance, if we are in the location where the pump is running and we get the command to switch the pump on. Then we do not need to get to the other location, since the pump is already running and we are in the correct location. Vice versa holds for the case, when the pump is turned off and we get the command to switch it off. But if we receive a command, which causes a state change of the pump, we must take one of the *action_ synch* transitions, which connects the two locations with each other. Assume, the pump is running and after taking the *read_ synch* transition, the command is to turn the pump off. Then we must leave our current location and get to the location off_1 that represents that the pump is not running. When we take the corresponding *action_ synch* transition, the check- back variable for the pump is set to *false*. This indicates that the command was executed and the pump is now turned off. Again vice versa hold for the other case.

The synchronization, the *control_ panel* and the *tank1* automaton are now given to SpaceEx. It should check if the given system only allows a water level between two and ten. Therefore it firstly builds the composition of the automata. We checked this composition and compared it to the one, which was build by our *CompositionCreator* which is depicted in figure 5.5. For a better readability we removed all activities from the locations, which just determine, that the variables do not change. These are the activities for the control variables, the check- back variable for the pump and the variables, which represent commands for the pump. The resulting automata are equal and both correct. After the analysis SpaceEx gives us a counter example. Upon this example the SFC verification tool decides to choose the conditional ODE system $(chkb\_P1\_on : \dot{h_1} = 0)$ for extending the SFC to an HSFC. The hybrid automata are now built again with modified informations. The synchronization automaton as well the *control_ panel* automaton are the same as before, since their behavior is not influenced by the extension of the SFC. This holds not for the automaton *tank1*. The conditional ODE system is assigned to the step on_1 in the SFC. As consequence of this, we add the condition $(chkb\_P1\_on == 1)$ to the invariant of the corresponding location in the automaton *tank1*. Also the activities of the location change. They are extended by the ODE $(\dot{h_1} = 0)$. Now we can stay in this location, only if the pump is running and meanwhile the water level of the tanks does not change. But this is not the only change derived

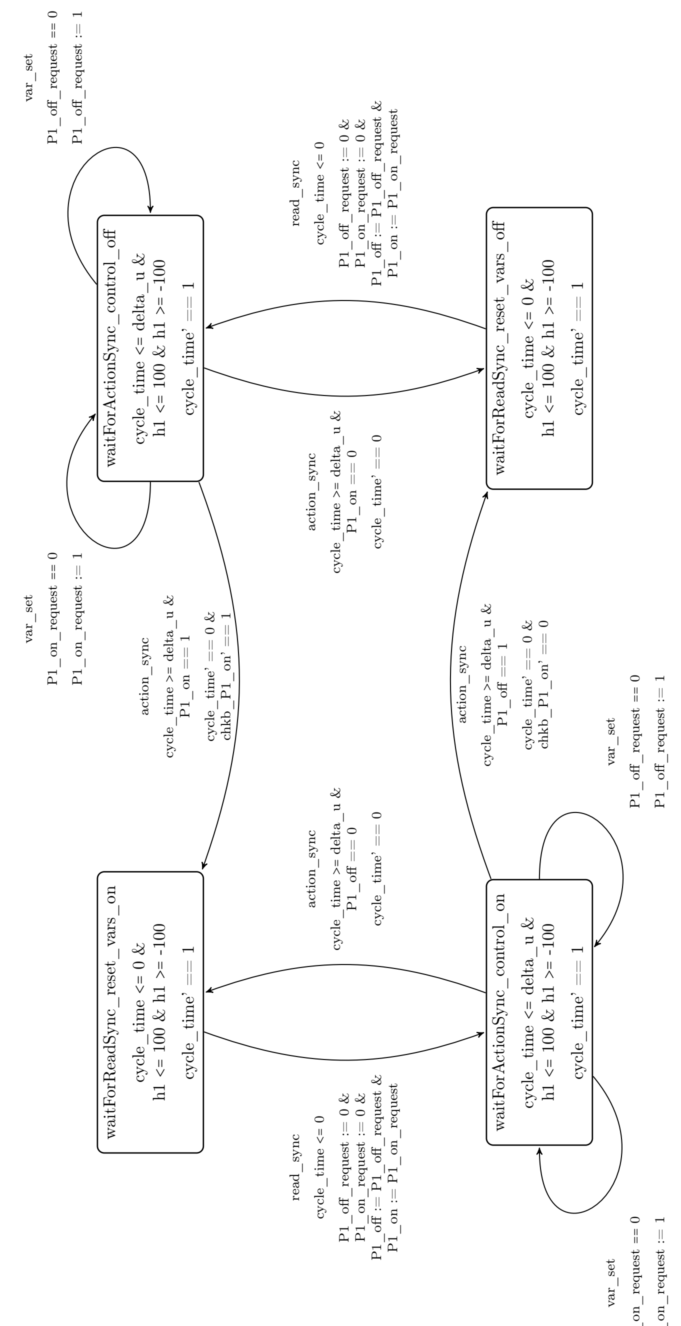


Figure 5.5: Composition ((synchronizer || control_panel) || tank1)

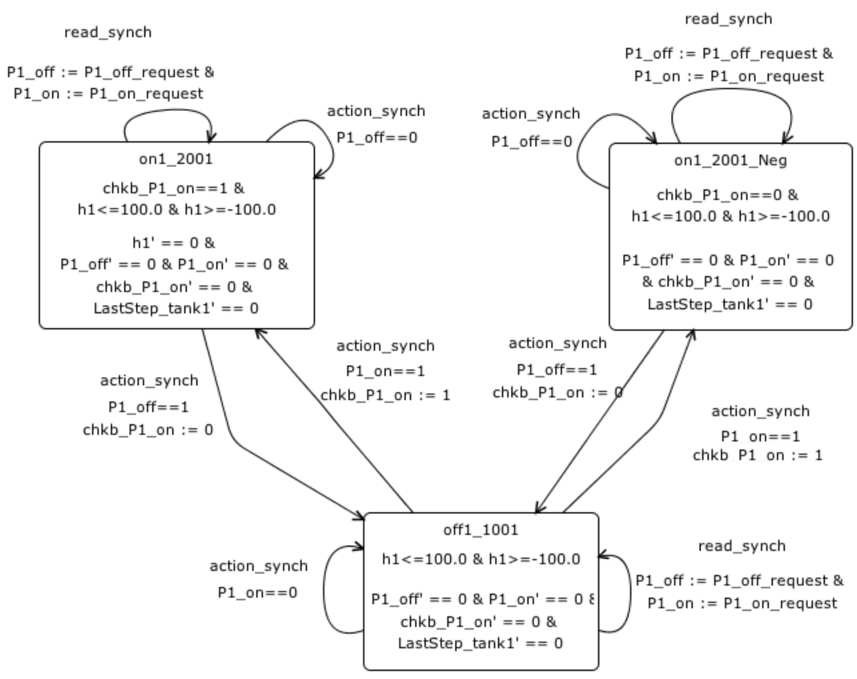


Figure 5.6: Hybrid automaton *tank1* extended with conditional ODE system

from the newly added conditional ODE system. To complete the transformation from the HSFC to an automaton, we have to add a new location. Therefore we take the original of the location we just extended and copy it. Before we insert it in the automaton, we expand its invariant by the negation of the condition, thus ($chkb\_P1\_on == 0$). We connect it to the other locations, by copying all incoming and outgoing transitions of the original location and assign them to the new location. The resulting hybrid automaton is shown in figure 5.6.

We pass the new set of automata again over to SpaceEx and let it give us the composition it build from the automata. This time we discovered some differences between the composition of SpaceEx and the one we get from our *CompositionCreator*.

Figure 5.7 and figure 5.8 show the part of composition, that differs. For a better readability we removed all activities from the locations, which just determine, that the variables do not change. Also we omitted each assignment, which determines, that the variable keep their value when a transition is taken in the composition built by SpaceEx.

The difference between the two compositions is a missing location in the composition build by SpaceEx. The *CompositionCreator* adds the location, which is made up from the location *waitForActionSynch* from the synchronization automaton, the location *control* from the *control_panel* and *on1_neg* by *tank1*, while SpaceEx omits especially this location. But although the composition from SpaceEx has one location less and because of this all incoming and outgoing transitions of the location do not exist, this has no effect on the behavior of the automaton. The reason for this is, that the location, that has been omitted by SpaceEx, can never be reached during the execution of the automaton. In figure

5.8 can be seen, that the only incoming transition to this location is a *read_ synch* transition from the location *waitForReadSynch_ reset_ vars_ on1_ neg*. This location can also never be entered. A part of it is retrieved from the location *on1_ neg* with the negated condition ($chkb\_P1\_on == 0$) we added during the refinement. So the location would only be entered, if we executed a command to switch the pump off. But since we copied all incoming and outgoing transitions from the location *on1* to connect the location with rest of the automaton *tank1*, the only incoming transition for the location in the composition is an *action_ synch* transition. For taking it we must get the command to switch the pump on and set the check- back variable to *true*. But this is exactly contrary to the valuation, which allowed be the invariant of the location. Thus, this location may never be entered for the reason, that the invariant can never be fulfilled. And since the location *waitForActionSynch_ control_ on1_ neg* can be entered only via this location, it can also never be reached.

SpaceEx performs a reachability analysis during the building of its composition. While our *CompositionCreator* only removes location in case of *false* as invariant or if there no incoming transition from other locations, it does a bit more. It takes the initial location and build all locations of the composition, that can be reached from it via transitions. After adding them, checks if they can be reached or if the assignments from the incoming transitions violates the invariant. If they do, SpaceEx not need to take care of this location, because it can never be entered.

So omitting a non reachable location has no effect on the behavior of the automaton and as consequence of it, we get equal results for the analysis of both compositions.

The advantage of the way SpaceEx builds the composition is, that only the reachable part of the composition is computed, which keep it as small as possible. But it can also happen, that the composition is not complete and parts of it are missing which influence the behavior of the whole system. A example for this is shown in section 5.2.2. Another disadvantage is, if the analysis fails, SpaceEx is not able to provide a composition anyway.

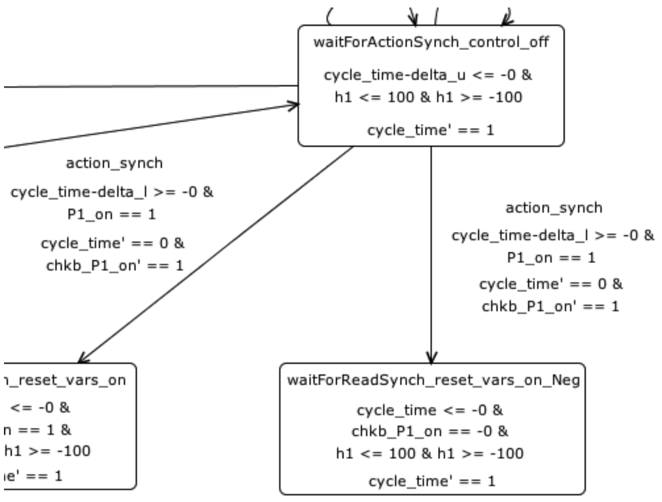


Figure 5.7: SpaceEx

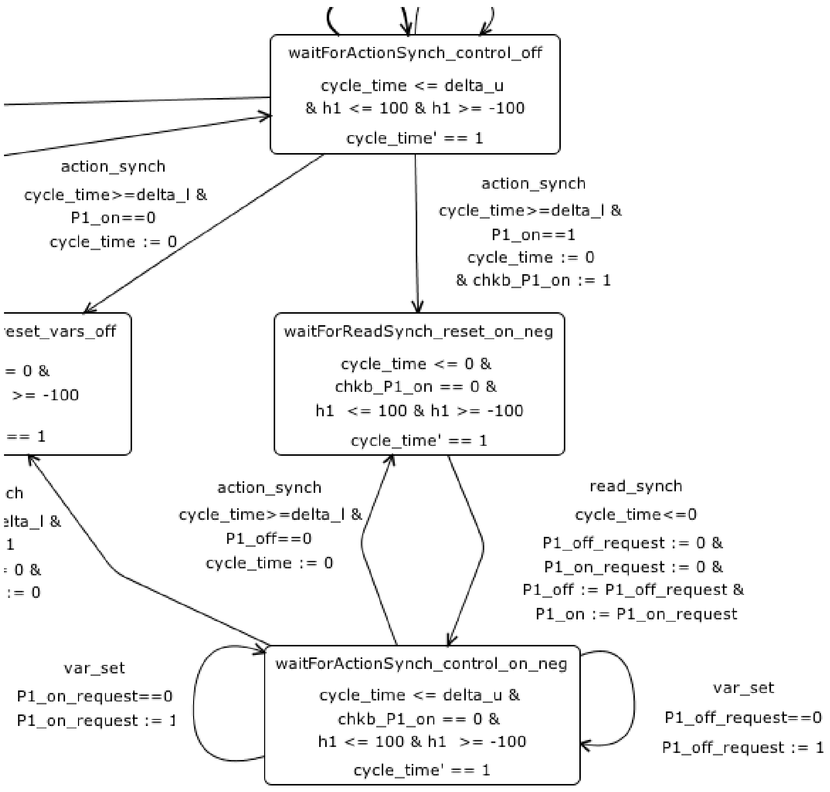


Figure 5.8: *CompositionCreator*

5.2.2 System with two tanks

Upon the informations of the system with two tank the SFC verification tool builds four automata for the given system. The automata *tank1* and *tank2* simulates the SFC for the tanks without any conditional ODE system assigned yet. They are both equal to automaton *tank1* from the first test case, shown in figure 5.4. Only the *read_ synch* transitions have one more assignments. They synchronize the check- back variable *chbk_ P1_ on* respectively *chkb_ P2_ on*. To know the state of the pump, which belongs to the other tank, is important for the later extension with ODE systems. As described in 5.1.2 the conditions of the ODE systems make use of the states of both pumps to be able to determine, how the water level of each tank evolves. To simulate the user input, we have again an automaton *controller_panel*, which sets the variables for the commands to wether switch a pump on or off randomly. These commands are given for each pump. So we have four *var_ set* transitions, for each pump one to switch it on and one to switch it off. The last one is the synchronizer automaton, which simulates the synchronization of the input and output variables at the beginning and at the end of every PLC cycle. It is added as *plc_synchronizer*. The SFC verification tool builds the input files for SpaceEx and starts the analysis. SpaceEx builds now the parallel composition of these automata $((plc\_synchronizer||controller\_panel)||tank2)||tank1$.

While we checked the composition, we found out, that it was not build correctly. The failure is caused by the outgoing transitions of the location, which is build of the following original locations: *waitForActionSynch* from *plc_synchronizer*, *control* from *controller_panel*, *on1* from *tank1* and *on1* from *tank2*.

The part of the composition, which is is build wrong is shown in figure 5.9. For a better comparison figure 5.10 shows also, how this part is build by the *CompositionCreator*. For a better readability we omit the activities, which determine, that control variables, check- back variables of the pumps as well as the requests for the commands to switch the pumps on or off may not be changed by a time step. We omitted also each assignment, which determines, that the variable keep their value when a transition is taken in the composition built by SpaceEx.

The location *waitForActionSynch_ control_ on1_ on1* with its invariant and activities is build correctly.

The last two location indicate, that both pumps are running, while we are in this location. But since *control* from *controller_panel* is also one of the original locations, we should be able to set one of the four control variables. The setting of these variables should simulate a random user input for switching the pumps off or leave them running. To realize this there should be four self loop transitions, two for each pump to switch them independently on or off. These transitions are missing in the composition of SpaceEx.

The outgoing *action_ synch* transitions are also missing. But *waitForAction–Synch* from the *plc_synchronizer* is an original location. Thus, normally we should wait for the time to pass by until the cycle time reaches at least δ_l and leave this location at latest, when the cycle time reaches δ_u. However, since there are no outgoing transitions, especially none with the label *action_ synch*, which should lead to another location, we are not able to go anywhere.

So we are in a location, where we should be able to simulate random user input and leave it afterwards. But we can do nothing of this and if we enter this location once, we can only let time elapse until the cycle time violates the invariant and the run of the automaton fails.

Since the composition is build wrong, the result of the analysis is not reliable. It gives us no informations about the wrong set up location and a counter example where the two pumps can never be switched on at the same time. All further analysis and refinement step are useless, because they rely on the wrong result of this incorrectly built composition.

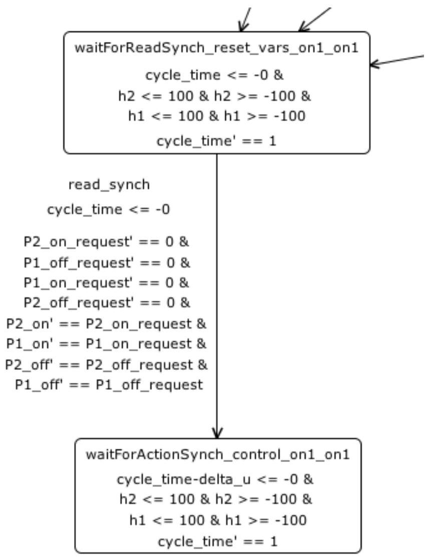


Figure 5.9: SpaceEx

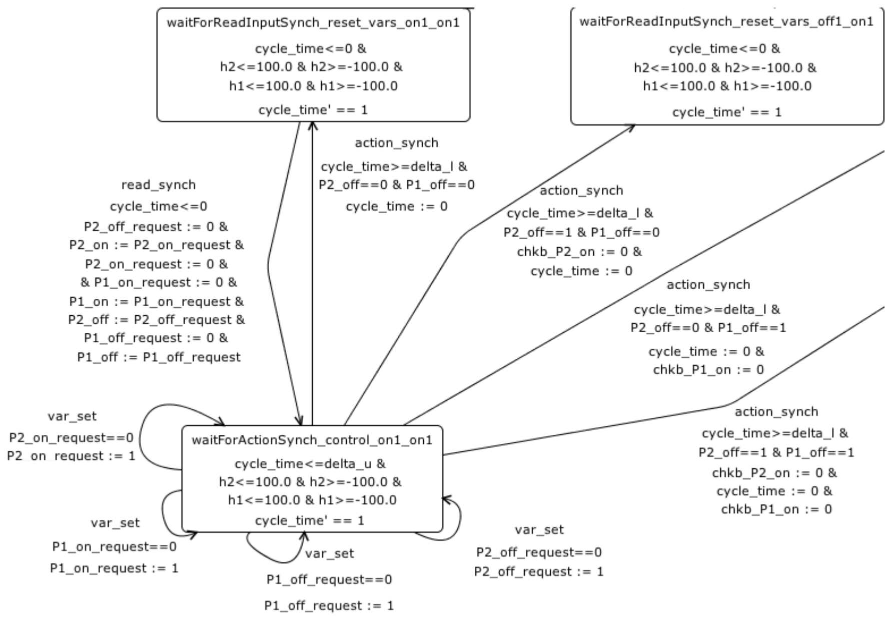


Figure 5.10: *CompositionCreator*

Chapter 6

Conclusion

6.1 Summary

We wanted to make the SFC verification tool as flexible as possible for the integration of other analysis tools. Previously the SFC verification tool was bounded to SpaceEx. During the transformation from (H)SFCs to hybrid automata the restrictions of SpaceEx were applied directly and in the end the automata were written into an input file for SpaceEx without saving them.

We introduced a new data structure for hybrid automata. Besides we implemented some *Creator* classes which transforms the (H)SFCs into hybrid automata without any restriction from any tool and save the automata in the data structure. Thus, we obtain general automata that are independent of any tool syntax. We are now able to apply any changes or restrictions needed for an analysis tool on the stored automata by our toolchain afterwards. Since some analysis tool can not handle several automata that run in parallel, we extended the toolchain with a component *CompositionCreator* that provides the composition of all automata in the system.

Hence, the integration of any analysis tool is possible now, and we are no longer bounded to SpaceEx. In our test cases we detected, that SpaceEx does not always compute the composition correct and thus, delivers us sometimes wrong results. This fact is very important, because a wrong result for the hybrid automata leads to a incorrect result for the safety verification of the SFCs. To avoid this, we compute the composition with our *CompositionCreator* and let SpaceEx only do the analysis.

6.2 Future work

As consequence of our results we plan to integrate other analysis tools such as $flow^*$ [12] to compute the reachable states. For each new tool we need to extend the toolchain, so it can handle the restriction for hybrid automata given by the tool. We have to implement a component for writing the input files for the new tool and one to read in the file with the result from the analysis needed for the refinement.

Since we try to keep our models as small as possible, we will improve the building of the composition further. Therefore we want extend the minimization of the intersection for the invariants and the guards. Until now no contrary equations such as $(x \leq 2 \wedge x \geq 3)$ are detected in the formulae, which prevent, that the formulae can ever be fulfilled. Locations or transitions with inconsistent invariants or guards can never be entered or taken, so omitting them has no effect on the behavior of the composition.

Bibliography

[1] Int. Electrotechnical Commission. Programmable controllers, part 3. *Programming Languages, 61131-3.*, 2003.

[2] Johanna Nellen and Erika Ábrahám. Hybrid sequential function charts. *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, 2012.

[3] G. Frehse, C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler. Spaceex: Scalable verification of hybrid systems. In *23rd International Conference on Computer Aided Verification (CAV)*, LNCS, 2011.

[4] K. Driessen. *Counterexample-Guided Abstraction Refinement for Hybrid SFC Verification*. RWTH Aachen, 2012.

[5] N. Bauer, R. Huuck, B. Lukoschus, and S. Engell. A unifying semantics for sequential function charts. In *LNCS*, 2004.

[6] E. Abráhám. Modeling and analysis of hybrid systems. Lecture Notes, 2012.

[7] T. A. Henzinger. The theory of hybrid automata. In *Eleventh Annual IEEE Symposium on Logic in Computer Science*, 1996.

[8] PLCopen Technical Committee 6. Xml formats for iec 61131-3. Technical report version 2.01, PLCopen, http://www.plcopen.org/pages/tc6_xml/, 2009.

[9] Beremiz. http://www.beremiz.org.

[10] G. Frehse. Phaver: algorithmic verifcation of hybrid systems past hytech. In *International Journal on Software Tools for Technology Transfer*, volume 10, 2008.

[11] C. Le Guernic and A. Girard. Reachabilityanalysis of linear systems using supportfunctions. In *Nonlinear Analysis: Hybrid Systems*, volume 4, 2010.

[12] X. Chen, E. Abráhám, and S. Sankaranarayanan. Flow*: An analyzer for non-linear hybrid systems. *Computer Aided Verification*, 2013.