hybr i2d Theory of Hybrid Systems Informatik 2

RWTH AACHEN

RWTH Aachen University
Rheinisch-Westfälische Technische Hochschule Aachen

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

Bachelor Thesis

MEMORY- AND TIME-RELATED ACTION QUALIFIERS IN HSFCs

Thomas Osterland
Matriculation Number: 297271

- December 2012 -

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

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.

Thomas Osterland
Aachen, den 22. April 2013

Contents

Chapter 1

Introduction

Sequential function charts (SFC) are a graphical programming language frequently used in combination with programmable logic controllers (PLC). Assembly lines, technical systems and power plants are realized by dozens of inter-operating PLCs controlling chemical, technical and atomic processes. It is crucial that these processes are safe in the manner that nothing miscellaneous will happen. To ensure this the *verification* of SFCs is an important research topic. A special challenge is the verification of SFCs considering its application context. That means the behavior of e.g. the plant or factory in which the SFC (on the PLC) is applied.
There exist several approaches to verify SFCs under consideration of the dynamic behavior of the plant [HKD98, BCMP]. Usually the plant is completely modeled by a *hybrid automaton.* In the verification process, the dynamic behavior of the automaton can be combined with the discrete behavior of the SFCs to check the reactions on dynamically changing environment conditions.
It is a difficult task to model a complete chemical plant and in general the resulting automaton will be large, such that verification algorithms will have difficulties in analyzing those systems due to the state space explosion and it will be annoying to restart the verification in case of an error in the model.
As described in [NÁ12] we avoid the creation of a complete plant model and represent its dynamic behavior by *conditional ordinary differential equation (ODE) systems*, which we introduce in the preliminaries in Chapter 2. Such conditional ODE systems can be added to steps of the SFC, which we introduce in Section 2.2.1, resulting in a *hybrid SFC* (HSFC). The following transformation into a hybrid automaton and a *Counter Example-Guided Abstraction Refinement* (CEGAR) based on reachability analysis can be used to verify an SFC in its plant context under consideration of different environment settings. Advantages are the smaller state space in comparison to approaches, which build the complete model of a plant and the ability to test the SFC with dedicated settings. That means it is possible to model the dynamic plant behavior arbitrarily abstract what can be useful to locate constitutional errors or to allow fast analysis due to small models.
The idea of the CEGAR approach is to iteratively refine an assignment of conditional ODE systems to steps of an SFC (Figure 1.1). On a given SFC with a specific assignment of conditional ODE systems we do a reachability test for a set of forbidden states. If we can reach one or more of them we get a counterexample, which describes a run through the automaton, where (for spurious counterexamples) at least one ODE is not satisfied. In this case additional dynamic behavior is added along this run, what means, that we change the assignment of conditional ODE systems to steps of the SFC and a new iteration of the reachability analysis is started. We do this until we either find a correct assignment or until the model provides a concrete counterexample.

Currently, the approach in [NÁ12] supports only a restricted syntax for SFCs. So parallel branching, nested SFCs and memory-/time-related action qualifier, which we introduce in the preliminary Section 2.2.1, are not supported by HSFCs. It is part of this bachelor thesis to extend the syntax and semantics of HSFCs

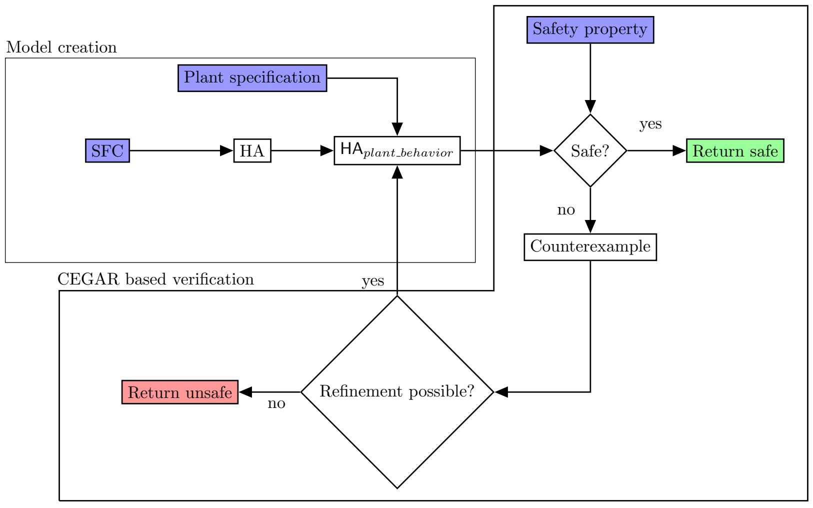


Figure 1.1: CEGAR based refinement.

to memory- and time-related action qualifiers and to give formal instructions for the transformation of those adapted HSFCs to hybrid automata.
Figure 1.2 illustrates our approach. Based on the transformation of (H)SFCs into hybrid automata and their extension to conditional ODE systems described in [NÁ12] (upper branch in Figure 1.2), we begin by giving transformation instruction for (H)SFCs, which contain memory-related action qualifiers (the lower branch in Figure 1.2) and in the following the transformation under consideration of conditional ODE systems. Using memory-related action qualifiers we are able to set and reset actions independent of any step activity. That means actions, which are always mapped to steps of an SFC (Section 2.2.1), are in general only executed if its corresponding step is active. Using memory-related action qualifiers, we can execute an action independent of the step activity.
Next we introduce the syntax and the semantics of SFCs and the transformation of time-related action qualifiers (central branch in Figure 1.2). This allows to execute actions for a specific duration or after a delay. In this way we are able to increase the potential of executing actions of an (H)SFC.
Finally we will analyze our concepts with respect to space complexity to indicate the practical usability.

This thesis is structured as follows: In Section 2 we give some preliminary definitions, which we need afterwards to define the syntax and semantics as well as the transformation of memory- and time-related (H)SFCs. In Section 3 we introduce memory-related action qualifiers. We explain their characteristics and define the syntax and semantics. A transformation of memory qualifier afflicted SFCs into hybrid automata and its complexity analysis is given at the end of this chapter. Section 4 addresses the timed action qualifiers. Also in this case we begin with an explanation of the characteristics and continue with the definition of its syntax and semantics. As in the chapter before we close by giving transformation instructions and a complexity analysis. We provide a practical example in Section 6 to demonstrate the transformation instructions,

$$\begin{tikzcd}
 & & & & \text{HA} + \text{ODE} \\
{} \arrow[r] & \text{SFC} \arrow[r] \arrow[ddrr] & \text{HA} \arrow[r] \arrow[urr] & \text{HA}_{Timed} \arrow[r] \arrow[d] & \text{HA}_{Timed} + \text{ODE} \\
 & & & \text{HA}_{Set+Timed} \arrow[r] & \text{HA}_{Set+Timed} + \text{ODE} \\
 & & & \text{HA}_{Set} \arrow[r] \arrow[u] & \text{HA}_{Set} + \text{ODE}
\end{tikzcd}$$

Figure 1.2: The transformation overview.

introduced in the foregoing chapters. At the end we give a conclusion of our results in Section 7.

Chapter 2
Preliminaries

In this chapter we introduce some basic principles. We begin by giving a benchmark, which is used to explain the following concepts and definitions. We continue by defining SFCs and HSFCs. Thereby we give intuitive explanations and present the formal syntax and semantics. Afterwards we continue by introducing hybrid automata, and give formal definitions for the transformation of SFCs and HSFCs into those, automata.

2.1 Example plant

Throughout the paper, we use the following example plant to give a demonstrative impression of transforming set-, reset- and time-qualified SFCs into hybrid automata. It is relatively small, but allows the understanding of fundamental ideas. We provide a second, more complex example in Section 6 to demonstrate the transformation in a more realistic context.

2.1.1 Simple mixer model

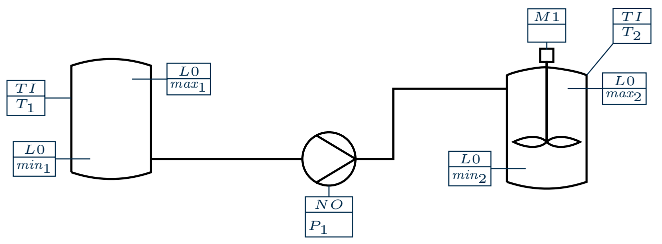


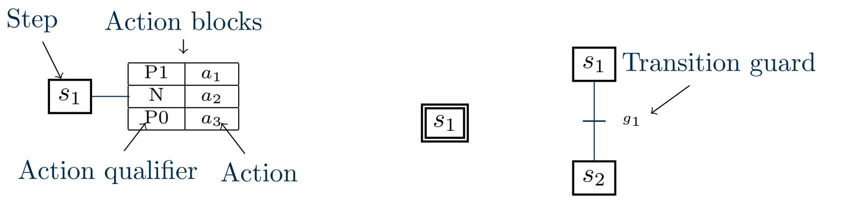
Figure 2.1: Simple mixer model.

The simple mixer model (Figure 2.1) consists of two tanks T_1 and T_2. While T_1 provides a liquid, T_2 contains a mixer M_1 to mix the incoming fluid. A pump P_1 is able to decant the liquid from T_1 into T_2.
The tanks are equipped with sensors min_1, min_2 and max_1, max_2. max_i is true, if the tank T_i is completely filled and min_i is false, if it is empty. If min_i is true and max_i false, the waterlevel of T_i is within the allowed bounds.
We introduce a variable *mixer*, which is true, if the mixer M_1 is running, otherwise it is false. In the same way, to determine the sensor states, we introduce variables $minj, maxj,\ j \in \{1, 2\}$. The mixer M_1 allows us to give intuitive examples for the usage of set/reset and timed action qualifiers, e.g., we can switch the mixer on after a given time delay.

2.2 Sequential Function Charts

Sequential Function Charts (SFCs) are a graphical programming language often used to program PLCs (Programmable Logic Controllers). They are originally introduced in the industry standard 61131 [Int03]. Because of ambiguous and not fully specified parts in the SFCs semantics, a first enhancement was given by [BHLE04] and further refined in [Bau04]. We follow in our definition mainly [NÁ12], which bases on the aboved mentioned contributions.

2.2.1 SFC syntax



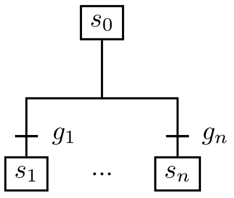
(a) Step and actions. (b) Initial step. (c) Transition with guard.

Figure 2.2: Example SFC.

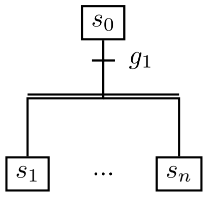
An SFC consists of a set Steps of *steps*, which are graphically represented as rectangular boxes commonly labeled with an identifier (Figure 2.2a). Each SFC has exactly one initial step $s_0 \in \mathit{Steps}$, in which the execution starts, i.e., which is active at the beginning (Figure 2.2b).
A set of *action blocks* is allocated to each step (Figure 2.2a). This set can also be empty. The union of all action blocks in the SFC is called B_{Act}. Action blocks $b = (q, a)$ are tuples containing the *action qualifier* $q \in \{P1, N, P0\}$ and the *action* $a \in \mathit{Act}$, where Act is the set of Actions. In [NÁ12] the qualifier set is restricted to **N**, **P1** and **P0**, but it is part of this thesis to extend this set.
The time an action is executed is specified by the action qualifier: An *entry* (P1) qualified action is executed only once if a step becomes active. In contrast to this an *exit* (P0) qualified action is executed only once if the assigned step becomes inactive and *do* (N) qualified actions are executed as long as the assigned step is active. To determine the execution order of the step allocated actions, an *action priority* $\sqsubset$ is given. The relation $a_1 \sqsubset a_2$ means that action a_1 is executed before action a_2.

An action can be an *assignment* of a value to a variable or an *SFC*. In the latter case we call them *nested* or *child* SFCs. Furthermore, an SFC maintains a set of typed *variables* Var, which can be grouped into input (Var_{input}), output (Var_{output}) and local (Var_{local}) variables. It is also possible that a variable is an input as well as an output variable. The standard [Int03] provides different variable types, e.g., int, real, bool, string, time and date.
Let us call the union of all variable type domains D. A type-preserving function $\sigma : \mathit{Var} \to D$, which assigns to each variable a value from its domain, is called a *state*. Let Σ be the set of all states. Furthermore, a *state transformation* is a function $f : \Sigma \to \Sigma$. The set of all state transformations is denoted by F.

Steps can be connected via *transitions* (Figure 2.2c), that means each transition has a set of *source* and a set of *target* steps. A transition begins at the bottoms of its source steps and ends at the tops of its target steps. This represents the execution direction. Regarding the target steps we distinguish two cases, which we call *branching* and *parallel branching*. The first is a short form for conditional branching (Figure 2.3a), while in the latter case all target



(a) Conditional branching.



(b) Parallel branching.

Figure 2.3: Different branch types.

steps are simultaneously activated (Figure 2.3b) and the actions of the target steps are executed in parallel. Parallel branching must eventually end into a single path, that means we can join parallel branchings in a step.
Transitions are *guarded* (Figure 2.2c). That means they can only be taken, if all source steps are active and the system state satisfies the guard. Guards are boolean expressions. We say that a guard g is satisfied by a state σ written $\sigma \models g$ if the variable valuation of σ fulfills g. We denote the set of all guards over the SFC variables by G^{Var}.
Transitions are *urgent*, that means if the guard is satisfied by a state, the transition must be taken immediately. If a step has several outgoing transitions with satisfied guards, a transition priority $\prec$ determines which transition will be taken. The relation $t_1 \prec t_2$ means that we take the transition t_1 before transition t_2. The set of all guarded transitions is denoted with *Trans*.
In [NÁ12], we consider only SFC, which do not have parallel branching. So we define $\mathit{source}(t) := s_i$ and $\mathit{target}(t) := s_j$ with $t = (s_i, g_i, s_j) \in \mathit{Trans}$ to get the source and target steps of a given transition t. We extend this notation to sets by defining $\mathit{source}(T) = \bigcup_{t \in T} \mathit{source}(t)$ and $\mathit{target}(T) = \bigcup_{t \in T} \mathit{target}(t)$
At last we define the *history flag Hist.* If the flag is set for an SFC, the execution will be continued in the state in that it was left after its last activation. Otherwise the execution always starts in the initial step. We call an SFC, which is not an action of another SFC, a *root SFC*. The history flag of a root SFC is always set to zero.

Definition 1 (SFC) *A* Sequential Function Chart *(*SFC*) is a tuple* $C = (\mathit{Var}, \mathit{Steps}, \mathit{Act}, s_0, \mathit{Trans}, \mathit{Blocks}, \sqsubset, \prec, \mathit{Hist})$ *with*

- *Var: a finite set of variables,*
- *Steps: a finite set of steps,*
- *Act: a finite set of actions,*
- $s_0 \in \mathit{Steps}$ *the initial step,*
- $\mathit{Trans} \subseteq (2^{\mathit{Steps}} \backslash \{\emptyset\}) \times G^{\mathit{Var}} \times (2^{\mathit{Steps}} \backslash \{\emptyset\})$*: a finite set of transitions,*
- $\mathit{Blocks} : \mathit{Steps} \to 2^{B_{\mathit{Act}}}$*,*
- $\sqsubset \subseteq \mathit{Act} \times \mathit{Act}$*: a total order on the actions,*
- $\prec \subseteq \mathit{Trans} \times \mathit{Trans}$*: a partial order on the transitions and*
- $\mathit{Hist} \in \{0, 1\}$*: a history flag.*

In the following it will be useful to access resources of the root SFC and all its nested SFCs of all levels recursively.

Definition 2 (Nested SFCs) *Let $C = (Var, Steps_C, Act_C, s_{0_C}, Trans_C, Blocks_C, \sqsubset_C, \prec_C, Hist_C)$ be an SFC and $\{C_1, ..., C_n\}$ nested SFCs with $C_i = (Var, Steps_{C_i}, Act_{C_i}, s_{0_{C_i}}, Trans_{C_i}, Blocks_{C_i}, \sqsubset_{C_i}, \prec_{C_i}, Hist_{C_i})$ for $i \in \{1, ..., n\}$, which can be child of the root SFC or another child SFC. We define:*

> *$\overline{C} := \{C, C_1, ..., C_n\}$: $\overline{C}$ contains the root SFC and all its nested SFCs at all levels.*
>
> *$\overline{Steps} := Steps_C \cup Steps_{C_1} \cup ... \cup Steps_{C_n}$: $\overline{Steps}$ contains the steps of the root SFC and its nested SFCs at all levels.*
>
> *$\overline{Trans} := Trans_C \cup Trans_{C_1} \cup ... \cup Trans_{C_n}$: $\overline{Trans}$ contains the transitions of the root SFC and its nested SFCs at all levels.*
>
> *$\overline{Act} := Act_C \cup Act_{C_1} \cup ... \cup Act_{C_n}$: $\overline{Act}$ contains the actions of the root SFC and its nested SFCs at all levels.*

We will now recall the *simple mixer model* from Section 2.1.1.

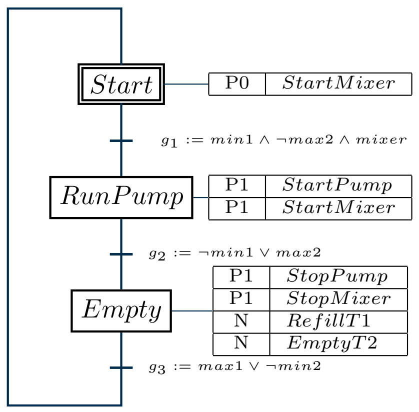

Start
P0 StartMixer
$g_1 := min1 \wedge \neg max2 \wedge mixer$
RunPump
P1 StartPump
P1 StartMixer
$g_2 := \neg min1 \vee max2$
Empty
P1 StopPump
P1 StopMixer
N RefillT1
N EmptyT2
$g_3 := max1 \vee \neg min2$

Figure 2.4: Simple mixer model SFC.

Let *StartMixer* be an action, which starts the mixer M_1 of tank T_2 and *StartPump* an action, which starts the pump P_1. Their reverse actions, which stop the mixer/pump are *StopMixer* and *StopPump*. We introduce two further actions, which allow to restore the initial situation. *RefillT1* refills the tank T_1 and *EmptyT2* drains tank T_2.
The SFC (Figure 2.4) models the following process: First the mixer M_1 of tank T_2 will be activated. If the mixer is running and T_2 is not full ($\neg max_2$) and T_1 is not empty (min_1), the pump P_1 will be activated in the next step. The fluid flows now from tank T_1 to tank T_2, until tank T_1 is empty ($\neg min_1$) or tank T_2 is full (max_2). Pump and mixer will be stopped, tank T_1 refilled and the tank T_2 decanted. The process can restart as soon as $max_1 \wedge \neg min_2$ holds.
Following Definition 1, we can describe the SFC of Figure 2.4 as follows:

Example 1 (SFC: Simple mixer model)
$SFC_{mixer} = (Var_{mixer},\ Steps_{mixer},\ Act_{mixer},\ Start,$
$Trans_{mixer},\ Blocks_{mixer},\ \sqsubset_{mixer},\ \emptyset,\ 0)$

- $Var_{mixer} := \{min1,\ max1,\ min2,\ max2,\ mixer\}$
- $Steps_{mixer} := \{Start,\ RunPump,\ Empty\}$
- $Act_{mixer} := \{StartMixer,\ StartPump,\ StopPump,$
  $StopMixer,\ RefillT1,\ EmptyT2\}$
- $Trans_{mixer} := \{(Start, g_1, RunPump), (RunPump, g_2, Empty),$
  $(Empty, g_3, Start)\}$
- $Blocks_{mixer} := \{Start \mapsto \{(P0, StartMixer)\},$
  $RunPump \mapsto \{(P1, StartPump), (P1, StartMixer)\},$
  $Empty \mapsto \{(P1, StopPump), (N, RefillT1),$
  $(P1, StopMixer), (N, EmptyT2)\}\}$
- $\sqsubset_{mixer} := StartPump\ \sqsubset\ StopPump\ \sqsubset\ RefillT1\ \sqsubset\ EmptyT2$
  $\sqsubset\ StartMixer\ \sqsubset\ StopMixer$

2.2.2 SFC semantics

Now we define the semantics of SFCs. Therefore we take a look at the work cycle of a PLC. The run of a PLC depends on the so called *cyclic scanning mode*. It can be divided into three single steps.

1. Read the input variables from the environment.
2. Execute the program on the stored input data.
3. Update the output with the computed values.

As in [NÁ12], we omit the first and the third step, where the communication with the environment is realized, and focus on the second step, where the SFC is executed. In case of SFC programmed PLCs - considered in this thesis -, we can subdivide the second step. A PLC executes an SFC by taking the SFCs transitions and applying the active actions with respect to the given order. After that it determines the active steps and actions for the next cycle. A characteristic of PLCs is that two cycles must not have the same runtime. Reasons for that are branches or loops, which can differ in their length. We follow [NÁ12] and define a lower bound δ_l and an upper bound δ_u for the execution time of one cycle.
We introduce the semantics given in [NÁ12] that is based on [Bau04]:
For an SFC $C = (Var, Steps, Act, s_0, Trans, Blocks, \sqsubset, \prec, Hist)$ we extend the system state (Section 2.2.1) by $(\sigma, readyS, activeS, activeA) \in \Sigma \times 2^{\overline{Steps}} \times 2^{\overline{Steps}} \times 2^{\overline{Act}}$, where σ is the current state and *readyS* is the set of steps, which are ready for activation. The set *activeS* contains the currently active steps and the sequence *activeA* the currently active actions. In contrast to *activeS*, *readyS* contains also the last active steps of inactive SFCs with true history flags.
We denote the set of all configurations by *Conf* and define the initial configuration to be $c_0 = (\sigma_0,\ \{s_0 \mid C_i \in \overline{C}, s_0\ initial\ step\ of\ C_i\},\ activeS_0,\ activeA_0)$, where σ_0 is the initial valuation assigning initial values to all variables: Boolean

variables are set to false and numerical variables are set to zero. The sets $activeS_0$ and $activeA_0$ are computed by executing the function *computeActiveSets* (Algorithm 1) with the initial mappings.

Let $c = (\sigma, readyS\ , activeS\ , activeA\ )$ be the current state of an SFC C. To compute the successor configuration, we first introduce the *enabled* and *taken* sets:

- $enabled(C, c) := \{(S, g, S') \in Trans(C) \mid S \subseteq activeS\ \wedge c \models g\}$
- $taken(C, c) :=$
  $\{t = (S, g, S') \in enabled(C, c) \mid \forall t_1 = (S_1, g_1, S`_1) \in enabled(C, c) :$
  $t \neq t_1 \rightarrow (t_1 \prec t \vee S \cap S_1 = \emptyset)\}$

The *enabled* function returns all transitions, which are enabled in the current cycle. That means all transitions with active source steps and satisfied transition guards. The *taken* function considers additionally the transition priority. Thereby it takes the set computed by the *enabled* function and removes all transitions, which may not be taken, since there exists a transition with a higher priority. Now we can compute the successor configuration:

Definition 3 (Transition relation) *The* transition relation $\rightarrow\subseteq Conf \times Conf$ *is defined as a configuration change. For a transition* $(c, c') \in\rightarrow$ *with* $c := (\sigma, readyS\ , activeS\ , activeA\ ) \rightarrow (\sigma', readyS', activeS', activeA') =: c'$*, we define*

- $readyS' = (readyS\ \setminus source(taken(C, c))) \cup target(taken(C, c))$.
- $(activeS', unsortedActiveA) =$
  $computeActiveSets(readyS',\ \emptyset,\ \emptyset,\ C,\ c, activeA\ \cap \overline{C})$
- $activeA' = sort(unsortedActiveA',\ \sqsubset)$.
- $\sigma' = f(\sigma) = (a_m \circ ... \circ a_1)(\sigma)$ *with* $activeA' = a_m \circ ... \circ a_1$ *and* $f \in F$.

The function computeActiveSets *is defined in Algorithm 1 and the function* sort *arranges the actions descending with respect to the given order* $\sqsubset$.

Algorithm 1 works as follows: First (line 1 - 4) the set of active steps for the currently regarded SFC C will be computed. We do this recursively for all nested SFCs (line 10-13). If the history flag was set or the SFC was already active in the cycle before, the current set of active steps can be computed by taking the intersection of the set *readyS'*, which contains in particular the last active steps of inactive SFCs[1], and the steps of the currently considered SFC C. If for an SFC the history flag is not set and it was not active in the cycle before the only active step will be the initial step s_0.
Next we compute the set of active actions (line 5-8) for the given SFC C. We first add the actions which were computed in the recursive calls of the function before and add the *exit* (**P0**) qualified actions from the steps which were left, the *entry* (**P1**) qualified actions of the steps which were entered, and the *do* (**N**) qualified actions of the active steps. Thereby we test whether the sequence of active actions $activeA'$ already contains the action. If this is the case the action

[1]with true history flag

Algorithm 1: computeActiveSets($readyS'$, $activeS$, $activeA$, C, c, $activeSFCs$)

```
   input  : $readyS'$, $activeS$, $activeA$,
            $C = (Var, Steps, Act, s_0, Trans, Blocks, \sqsubset, \prec, Hist)$,
            $c$, $activeSFCs$
   output: $activeS'$, $activeA'$
   /* Add the local active steps of C to $activeS'$.                    */
1  if $Hist = 1 \lor C \in activeSFCs$ then
2      $activeS' := activeS \cup (Steps \cap readyS')$;
3  else
4      $activeS' := activeS \cup \{s_0\}$;
   /* Collect the local active actions and their qualifiers.            */
5  $activeA' := activeA$;
6  foreach $s \in Steps, b = (q, a) \in Blocks(s)$ do
7      if $(q = P0 \land s \in source(taken(C, c))) \lor (q = P1 \land s \in$
       $target(taken(C, c))) \lor (q = N \land s \in activeS') \land a \notin activeA'$ then
8          $activeA' := activeA' \circ a$;
9  end
   /* Compute $activeA'$ and $activeS'$ for each active nested SFC     */
10 foreach $s \in Steps \cap activeS', b = (q, a) \in Blocks(s)$ do
11     if $a \in \overline{C}$ then
12         $(activeS', activeA') :=$
13         computeActiveSets($readyS', activeS', activeA', a, c, activeSFCs$);
14 end
15 return $(activeS', activeA')$;
```

will not be added twice. The reason for that is the following: If a step contains the same action multiple times (with the same or different qualifiers), we consider them as one action in the execution. We can simulate the **P** qualifier, which is also introduced in the IEC standard [Int03] and which will execute an action at the beginning and the ending of a step activation by adding the **P**-qualified action **P0**- and **P1**-qualified to the corresponding step.
Finally we compute the active steps and actions recursively for each active nested SFC (line 10 - 13).
In conclusion, Algorithm 1 computes all actions and their corresponding steps, which become active in the next cycle.

2.2.3 Additional semantic definitions for SFCs

We define some functions over SFCs, to enable more intuitive descriptions of definitions and examples. We begin by defining functions, which return the **P0**-, **N**- and **P1**-qualified actions of a step.

Definition 4 (Specific action set function) *Let* $C = (Var, Steps, Act, s_0, Trans, Blocks, \sqsubset, \prec, Hist) \in SFC$ *be an SFC and* $c = (\sigma, readyS, activeS, activeA) \in Conf$ *its current configuration. We define*

- $Entry : Steps \rightarrow 2^{Act}$ *with* $Entry(s) = \{a \mid (P1, a) \in Blocks(s)\}$
- $Do : Steps \rightarrow 2^{Act}$ *with* $Do(s) = \{a \mid (N, a) \in Blocks(s)\}$

- $Exit : Steps \to 2^{Act}$ *with* $Exit(s) = \{a \mid (P0, a) \in Blocks(s)\}$

$Entry(s)$, $Do(s)$ and $Exit(s)$ return the corresponding actions of a step. We introduce three further functions, which consider nested SFCs and parallel branching. Thereby we regard all steps, which are in the current state active.

- $Entry_{Nested} : Conf \to 2^{Act}$ *with* $Entry_{Nested}(c) = \bigcup_{s \in activeS} Entry(s)$
- $Do_{Nested} : Conf \to 2^{Act}$ *with* $Do_{Nested}(c) = \bigcup_{s \in activeS} Do(s)$
- $Exit_{Nested} : Conf \to 2^{Act}$ *with* $Exit_{Nested}(c) = \bigcup_{s \in activeS} Exit(s)$

2.3 Hybrid Sequential Function Charts

Hybrid Sequential Function Charts (*HSFCs*) are an extension of SFCs introduced by [NÁ12]. They allow to add a *conditional ordinary differential equation* (*ODE*) system to an SFC. In this way, the dynamic plant behavior can be integrated into an SFC by mapping the conditional ODE system to specific steps of the SFC. The result are hybrid SFCs, which can be used to verify an SFC regarding the dynamic behavior of the plant.

2.3.1 HSFC syntax

We start by defining *conditional ODE systems.* A conditional ODE system consists of a set of ordinary differential equations and a condition. The equations model specific plant behavior under the premise that the associated condition holds. An equation is defined over a set of continuous variables, which are, in contrast to a SFC, part of the HSFCs.

Definition 5 (Conditional ODE system) *Let Var be a set of variables, $Var_C \subseteq Var$ a set of continuous variables and ODE_{Var_C} be the set of all ordinary differential equations over the set Var_C. Let $Conds$ be the set of all conditions being quantifier-free linear real arithmetic formulas over the set Var (extended with boolean variables). We call a tuple $(cond : \ equ)$ with $cond \in Conds$ and $equ \subseteq ODE_{Var_C}$* conditional ODE system *and denote the set of all conditional ODE systems over Var_C by $CODE_{Var_C}$.*

These conditional ODE systems can be assigned to steps of an SFC, which we call HSFC in the following. We give some example conditional ODE systems to get an intuition and define them in a formal way accordingly.
Let us regard the simple mixer model (Section 2.1.1). We define the following conditional ODEs:

$$\neg\, min1 \lor max2 \lor \neg\, P_1 \lor \neg\, mixer : \ h_2' = 0, \ h_1' = 0 \tag{2.1}$$

$$min1 \land \neg\, max2 \land P_1 \land mixer : \ h_2' = c_2, \quad h_1' = -c_1 \tag{2.2}$$

$$\neg P_1 \lor \neg\, mixer : \ h_2' = 0, \ h_1' = 0 \tag{2.3}$$

The first conditional ODE (2.1) describes that if the tank T_1 is empty, the tank T_2 is full or the pump or the mixer are deactivated the water levels of tank T_1 and tank T_2 will not change. The second conditional ODE (2) has also two equations. If the tank T_1 is not empty and tank T_2 not full and mixer M_1 and pump P_1 are running, the water level of tank T_2 will increase with a value c_2 per

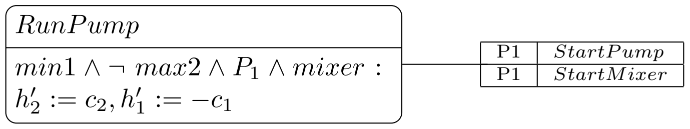

RunPump
$min1 \land \neg\ max2 \land P_1 \land mixer$:
$h_2' := c_2, h_1' := -c_1$
P1 | StartPump
P1 | StartMixer

Figure 2.5: Step *RunPump* with two assigned conditional ODEs.

time unit, while the water level of tank T_1 will decrease with speed c_1. The last conditional ODE (3) is a kind of safety criterion. Only if pump P_1 and mixer M_1 are running the water level may change.
A graphical representation of a HSFCs step is given in Figure 2.5. The conditional ODE system will be added to a differentiated part of the rectangle containing the label. That means we split each rectangle into two parts where the lower will be equipped with the conditional ODE system.

A second, more complex example can be considered in Section 6.
We recall the definition of HSCFs from [NÁ12].

Definition 6 (HSFC Syntax) *An HSFC is a tuple HSFC* $= (Var, Steps, Act, s_0, Trans, Blocks, Dyn, \sqsubset, \prec, Hist)$. *Steps, Act,* s_0*, Trans, Blocks,* $\sqsubset$ *and* $\prec$ *can be adopted from Definition 1 (SFC Syntax).*

- $Var := Var_{SFC} \cup Var_C$*, where* Var_{SFC} *is the variable set of Definition 1 and* Var_C *the set of continuous variable from above (Definition 5).*
- $Dyn : Steps \rightarrow CODE^*_{Var_C}$ *is a function, mapping a list of conditional ODEs* $(cond_1, equ_1), ..., (cond_{n_s}, equ_{n_s})$, $cond_i \in Conds$, $equ_i \subseteq ODE_{Var_C}, i \in \{1, ..., n_s\}, s \in Steps$ *to each step.*

Conditional ODE systems can be derived from the hardware specification and the plant environment. They allow the verification of an SFC under consideration of its application area.

2.3.2 HSFC semantic

In contrast to an SFC, an HSFC $C = (Var, Steps, Act, s_0, Trans, Blocks, Dyn, \sqsubset, \prec, Hist)$, needs a configuration tuple $c = (\sigma, readyS, activeS, activeA, activeD)$ $\in \Sigma \times 2^{Steps(C)} \times 2^{Steps(C)} \times 2^{Act} \times 2^{ODE_{Var_C}}$ with one additional set *activeD*, which contains the currently active *ODE*s. Active *ODE*s are the first *conditional ODE systems* of steps, which fulfill the *ODE* condition.
An *ODE* models dynamic, continuous parts, while an SFC follows a discrete, step behavior. This leads to an adaption of the SFC semantics, especially the *cyclic scanning mode* of Section 2.2.2. We add a fourth step - we will call it *time step* - in which we update the time advance and continuous variables. That means we measure the time elapse $\delta_l \leq t \leq \delta_u$ of each cycle and update the continuous variables according to the active *ODE*s and the time *t*.
We recall the definition of HSCF semantics from [NÁ12].

Definition 7 (Time steps) *We define the time step transition relation* $\rightarrow \subseteq Conf_{HSFC} \times Conf_{HSFC}$ *as follows:*
$(\sigma, readyS, activeS, activeA, activeD) \rightarrow (\sigma', readyS, activeS, activeA, activeD')$ *iff*

- *activeD'* :=
  $\bigcup_{s \in activeS} \{d \mid \exists\ (cond_i, O_i) \in Dyn(s) \wedge d \in O_i \wedge \sigma \models cond_i \wedge \bigwedge_{j=1}^{i-1} \sigma \not\models cond_j\}$
- $\sigma'(v) = \sigma(v)$ *for all* $v \notin Var_C$ *and* $\sigma'|_{Var_C} = f(t)$ *for* f *a solution of* *activeD* *with* $f(0) = \sigma|_{Var_C}$ *and* $\delta_l \leq t \leq \delta_u$.

2.4 Hybrid automata

We introduce *hybrid automata* following [ACH$^+$95] and explain the basic principles to use them in the transformation of an SFC.
A hybrid automaton (Figure 2.6) has a set of real-valued variables *Var* and a set of locations. We call a function $v \in V$, $v : Var \to \mathbb{R}$, which assigns to each variable a value, a *valuation* of *Var*. We write $v[x := 0]$ for the valuation v' with $v'(x) = 0$ and $v'(y) = v(y)$ for all $y \neq x$. We introduce a function $h : V \times Var \times \mathbb{R} \to V$ with $h(v, a, z) = v[a := z]$ and define the order $\sqsubset$ of the assignment executions in transitions. For a valuation function $v \in V$, values $z, z' \in \mathbb{R}$ and variables $a, b \in Var$ we write $h(v, a, z) \sqsubset h(v, b, z')$, if the assignment $h(v, a, z)$ will be executed before $h(v, b, z')$. In this case we execute the assignments as follows: $h(v, a, z) = v'$ and $h(v', b, z')$. We denote the set of all assignment functions with *Asg*. A *state* (l, v) of a hybrid automaton is a combination of a location $l \in Loc$ and a variable valuation $v \in 2^{\mathbb{R}}$.

Definition 8 (Hybrid automata) *A hybrid automaton is a nine-tuple* $H = (Loc, Var, Edge, Act, Inv, L, sync, Init, \sqsubset)$ *with:*

- *Loc: a finite set of locations,*
- *Var: a finite set of real variables,*
- $Edge \subseteq Loc \times 2^V \times 2^V \times Loc$*: a finite set of edges,*
- *Act: a function assigning time-invariant functions* $f : \mathbb{R}_{\geq 0} \to V$ *to a location,*
- $Inv : Loc \to 2^V$*: a function assigning an invariant to each location,*
- *L: a set of synchronization labels,*
- $sync : Edge \to L$*: a function assigning a synchronization label to an edge,*
- $Init \subseteq Loc \times V$*: a set of initial states and*
- $\sqsubset \subseteq Asg \times Asg$ *a total order on the assignment functions.*

There are two different transition types in hybrid automata. We distinguish *discrete state changes (jumps)* and *time delay (flows)*. The first represents a location change, while the second one models time elapse and changes the variable valuation according to actions *Act*, which are allocated to currently active locations. The location invariant must be always satisfied. A location cannot be entered if its invariant is not fulfilled. As in SFCs, transitions are guarded. A transition can only be taken if its guard is satisfied. In addition, a transition is able to assign values to variables. So we can manipulate the state by taking a transition.

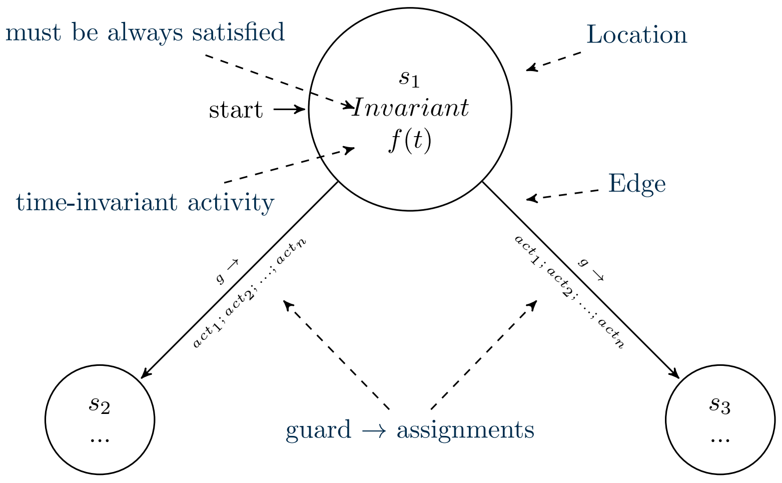


Figure 2.6: Example: Hybrid automaton.

2.5 Transformation to hybrid automaton

To enable the verification of hybrid sequential function charts, we give a transformation into hybrid automata. On hybrid automata we can apply available tools, which are not available for SFCs or other less formalized structures.
We start by recalling the transformation of an (H)SFC into a hybrid automaton as introduced in [NÁ12]. We do this in two steps, beginning with the transformation of an SFC and continue by extending this transformation to work on HSFCs (Figure 2.7).

$$\begin{tikzcd}
 & & & \text{HA} + \mathit{ODE} \\
\text{SFC} \arrow[r, red] \arrow[drr] & \text{HA} \arrow[r] \arrow[urr, red] & \text{HA}_{\mathit{Timed}} \arrow[r] \arrow[d, leftrightarrow] & \text{HA}_{\mathit{Timed}} + \mathit{ODE} \\
 & & \text{HA}_{\mathit{Set}} \arrow[r] & \text{HA}_{\mathit{Set}} + \mathit{ODE}
\end{tikzcd}$$

Figure 2.7: Overview of the transformations.

2.5.1 Transformation of an SFC

The method of [NÁ12] can be partitioned into three main ideas. The first describes the transformation of the SFC steps, especially the initial step. The second explains how to model the PLC induced time elapse and the third gives instructions for transferring the SFC transitions into the hybrid automata under consideration of formerly defined priorities.
We will begin with the introduction of the PLC induced time elapse.

Synchronization automaton

To model the *cyclic scanning mode* of a PLC, we recall our definitions of upper δ_u and lower δ_l bounds for the cycle time (Section 2.2.2). A state change must happen until δ_u but may not begin before δ_l time units has passed.
To handle this, we create a dedicated automaton - the synchronization automaton - which is used to synchronize the cycle times of several components (Figure 2.8).

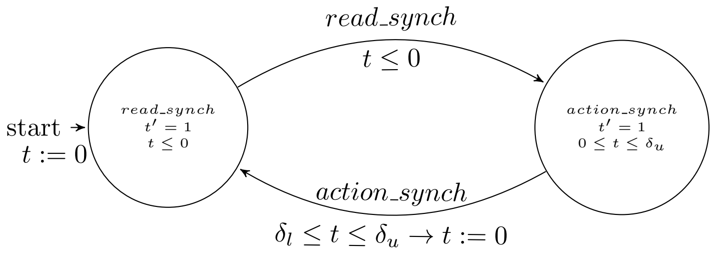


Figure 2.8: Synchronization automaton.

The automaton consists of two locations, which are connected via transitions. The first, initial location waits for the *read synchronization*. The synchronization takes place, if the transition to the next location will be taken. This happens immediately, because the location invariant $t \leq 0$ of the initial location becomes invalid by any time ellapse.
The transition from the initial step to the second step is labeled with *read_synch*. That means transitions in other automata, which are labeled with the same identifier, will be taken simultaneously.
In the synchronization automaton, we have a second transition leading from the the second location back to the initial location. It is labeled with the identifier *action_synch* and will be taken while the time t is within the lower bound δ_l and the upper bound δ_u of a PLC cycle.
The idea is to model the different steps of a PLC cycle in this way. Input values will be read at the beginning of each PLC cycle and may not change within the cycle. Therefore we take a *read_synch* transition, which allows to copy the current input values into other variables to fix their value for the duration of the cycle. The *action_synch* transition allows to execute actions in dependency to the read values, within the bounds of a PLC cycle. The synchronization label ensures that the corresponding transitions in related automata will be taken simultaneously and within the bounds. So we get unambiguous behavior in the hybrid automata, which is comparable to the behavior of a PLC executing an SFC.

The transformation

Steps are directly transformed to locations, that means every step in the SFC has an equivalent location in the hybrid automaton, especially the initial step of the SFC is equal to the initial step of the automaton. Since we model the cyclic scanning mode through the *synchronization automaton* running in parallel, we do not need to add invariants or activities to the locations of the hybrid automaton. But we will do this later on at the transformation of HSFCs.

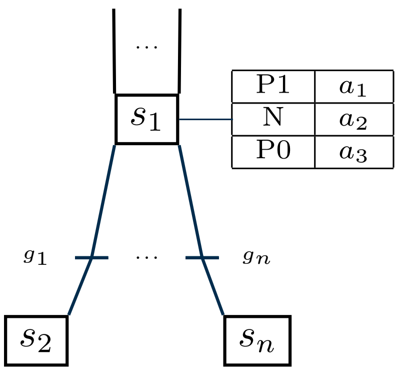
The action blocks of steps are modeled by adding the actions to the transitions in dependence to their action qualifier. For each transition in the SFC, we create a transition in the hybrid automaton, labeled with *action_sync*. Taking a transition the hybrid automaton executes the **P0**-qualified actions of the source and the **P1-** and **N**-qualified actions of the target step considering the defined order $\sqsubset$ (Figure 2.9). To realize the execution of **N** actions, each location in the hybrid automaton will be equipped with a self-loop, labeled with *action_sync* as well. If no outgoing transition guard is satisfied the self loop will be taken and the assigned **N** actions executed. The transition guard of the self-loop is the negated conjunction of all guards assigned to outgoing transitions.
The transition priority $\prec$ will be modeled by simply adding the SFC guards of the transitions to the automata transitions. For the transitions with lower priority the conjunction of the negations of the foregoing higher-priority transition guards are added as well. To realize the handling of input variables, each location will be also equipped with a *read_sync* labeled transition, which stores the input variables as described above.
We recall the transformation definition of [NÁ12] in the following:

Definition 9 (SFC $\rightarrow$ Hybrid automaton) *Let* $C = (Var, Steps, Act, s_0, Trans, Blocks, \sqsubset, \prec, Hist)$ *be an SFC without parallel branching and nested SFCs. Then its* transformed hybrid automaton *is* $H = (Loc, Var_H, Edge, Act_H, Inv, L, sync, Init, \sqsubset_H)$ *with:*

- $Loc := Steps$,
- $Var_H := Var \cup \{a_{global} \mid a \in Var_{input}\}$,
- $Edge := \cup_{s \in Steps} Edge_s$, *where for each step* $s \in Steps$ *with outgoing transitions* $t_1, ..., t_n \in Trans$ *and priority* $t_1 \succ ... \succ t_n$, $Edge_s := Edge_{ActionSync} \cup Loop_{ActionSync} \cup Loop_{ReadSync}$ *with*
  - $Edge_{ActionSync} := \{(s, \mu_i, s_i) \mid \exists i \in \{1, ..., n\} t_i = (s, g_i, s_i) \in Trans\}$, *labeled with action_sync and* $\mu_i := \{(v, v') \mid v \models g_i \wedge \bigwedge_{j=1}^{i-1} \neg g_j \wedge a_n \circ ... \circ a_1(v) = v'\}$, *with* $a_1 \sqsubset ... \sqsubset a_n$, $\{a_1, ..., a_n\} = Exit(s) \cup Entry(s_i) \cup Do(s_i)$
  - $Loop_{ActionSync} := \{(s, \mu, s) \mid s \in Steps\}$, *labeled with action_sync and* $\mu := \{(v, v') \mid v \models \bigwedge_{j=1}^{n} \neg g_j \wedge a_n \circ ... \circ a_1(v) = v'\}$, *with* $a_1 \sqsubset ... \sqsubset a_n$, $\{a_1, ..., a_n\} = Do(s)$
  - $Loop_{ReadSync} := \{(s, \mu, s) \mid s \in Steps\}$, *labeled with read_sync and* $\mu := \{(v, v') \mid v \mid_{Var} = v' \mid_{Var} \wedge \forall\, a \in Var:\ v'(a) = v'(a_{global})\}$,
- $Act_H(s) = \{f : \mathbb{R}_{\geq 0} \rightarrow V \mid \forall\, t \geq 0 :\ f(t) = f(0)\}$ *For all* $s \in Loc$ *and* $v' = 0,\ \forall\, v \in Var_H$,
- $Inv(s) = \emptyset$ *for all* $s \in Loc$,
- $L := \{action\_sync, read\_sync\}$,
- $sync : Edge \rightarrow L$ *with*
  $$L(t) = \begin{cases} action\_sync & \text{if } t \in Edge_{ActionSync} \cup Loop_{ActionSync} \\ read\_sync & \text{if } t \in Loop_{ReadSync} \end{cases},$$

- *Init* $:= \{(s_0, v_0)\}$, *where* v_0 *is the initial variable valuation of* Var_H *and*
- $\sqsubset_H \subseteq Asg \times Asg$ *with for all* $v \in V$ *and for all variables* $b = act_1^{-1}(\mathbb{R}), c = act_2^{-1}(\mathbb{R}) \in Act : h(v, b, act_1) \sqsubset_H h(v, c, act_2)$ *iff* $act_1 \sqsubset act_2$.

In the following we denote the assignment of global input variables $a := a_{global}, a \in Var_{input}$ *by* readInput *to allow understandable examples.*



(a) Step with actions and transitions.

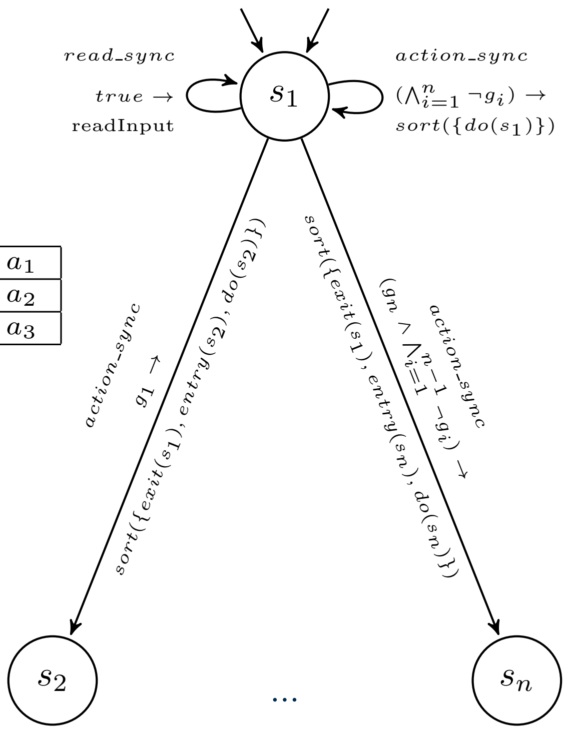

$read\_sync$
$true \rightarrow$
readInput
s_1
$action\_sync$
$(\bigwedge_{i=1}^{n} \neg g_i) \rightarrow$
$sort(\{do(s_1)\})$
$action\_sync$
$g_1 \rightarrow$
$sort(\{exit(s_1), entry(s_2), do(s_2)\})$
$action\_sync$
$(g_n \wedge \bigwedge_{i=1}^{n-1} \neg g_i) \rightarrow$
$sort(\{exit(s_1), entry(s_n), do(s_n)\})$
s_2
...
s_n
(b) Hybrid automaton transformation.

Figure 2.9: Transformation process: SFC $\rightarrow$ HA.

If we apply the transformation algorithm on the simple mixer benchmark, we get the hybrid automaton depicted in Figure 2.10.

2.5.2 Transformation of an HSFC

The transformation of an HSFC into a hybrid automaton is the extension of the transformation of an SFC, shown above (Section 2.5.1).
We have seen in Section 2.3.1 how we extend an SFC by a set of (conditional) ordinary differential equation systems to get an HSFC, which is able to model dynamic plant behavior. Since we already know how to translate an SFC into a hybrid automaton (Section 2.5.1), the challenge is to extend this method by adding ODEs to the resulting hybrid automaton.
For a location l in the hybrid automaton, which represents a step s in the corresponding SFC with allocated ODEs $Dyn(s) = (cond_1 : ODE_1, ...,$ $cond_n : ODE_n)$, we subdivide l into locations $l_1, ..., l_{n+1}$ (Figure 2.11). The

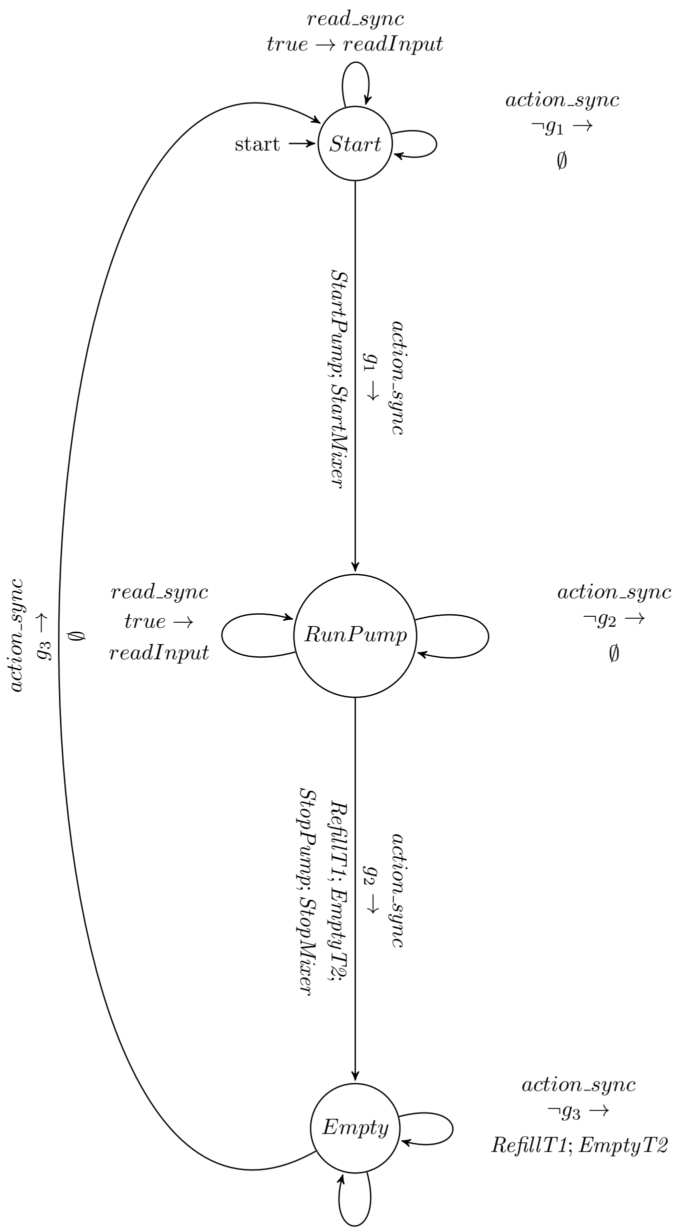


Figure 2.10: Simple mixer hybrid automaton.

condition $cond_i$, will be added to the invariant of location l_i and the equation ODE_i to its actions for $i \in \{1, ..., n\}$. The $n+1$'th location represents the case in which no condition holds and chaotic behavior is assumed for the continuous variables. We introduce a function *cl*, which replaces every $<, >$ with $\leq, \geq$. The reason is, that the invariants of the locations depends only on the context of the process and not on variables, which we can influence directly by transition assignments. So we avoid the case that a transition can never be taken because the invariants of the target steps becomes never valid.
The created locations are pairwise connected via transitions, which allows switching between ODEs depending on their condition. Thereby we introduce a variable x, which will be increased in each split location for every time unit and reset by transitions. So we eliminate Zeno behavior, what means to take infinitely many transitions in finite time.
We recall the definition of [NÁ12]:

Definition 10 (HSFC → Hybrid automaton) *Let $C = (Var, Steps, Act, s_0, Trans, Blocks, Dyn, \sqsubset, \prec, Hist)$ be an HSFC as defined in Definition 6. That means in particular without parallel branching and nested SFCs. We transform it into a hybrid automaton $H = (Loc, Var_H, Edge, Act_H, Inv, L, sync, Init, \sqsubset_H)$ with:*

- $Loc := \bigcup_{s \in Steps} Loc_s$
  $Loc_s := \{s_i \mid 1 \leq i \leq |Dyn(s)| + 1\}$
- $Var_H := Var \cup \{a_{global} \mid a \in Var_{input}\} \cup \{x\}$
- *Let $s \in Steps$ and $Dyn(s) = ((cond_1 : ODE_1), ..., (cond_m : ODE_m))$. If s has outgoing transitions $t_1, ..., t_n \in Trans$ with ordering $t_1 \succ ... \succ t_n$ then $Edge := Edge_{ActionSync} \cup Loop_{ActionSync} \cup Loop_{ReadSync} \cup Edge_{CopyTrans}$:*
  - $Edge_{ActionSync} := \{(s_j, \mu, s^i_k) \mid s_j \in Loc_s,\ s^i_k \in Loc_{s^i},\ (s, g_i, s^i) = t_i \in Trans\}$, *labeled with action_sync and*
    $\mu := \{(v, v') \mid v \models g_i \wedge \bigwedge_{j=1}^{i-1} \neg g_j \wedge a_n \circ ... \circ a_1(v) = v'\}$ *with* $a_1 \sqsubset ... \sqsubset a_n$, $\{a_1, ..., a_n\} = Exit(s) \cup Entry(s^i) \cup Do(s^i)$
  - $Loop_{ActionSync} := \{(s_k, \mu, s_k) \mid s_k \in Loc_s,\ s \in Steps\}$, *labeled with action_sync and*
    $\mu := \{(v, v') \mid v \models \bigwedge_{j=1}^{n} \neg g_j \wedge a_n \circ ... \circ a_1(v) = v'\}$ *with* $a_1 \sqsubset ... \sqsubset a_n$, $\{a_1, ..., a_n\} = Do(s)$
  - $Loop_{ReadSync} := \{(s, \mu, s) \mid s \in Steps\}$, *labeled with read_sync and*
    $\mu := \{(v, v') \mid v|_{Var} = v'|_{Var} \wedge \forall\, a \in Var:\ v'(a) = v'(a_{global})\}$,
  - $Edge_{CopyTrans} := \{(s_k, \mu, s_l) \mid k \neq l,\ s_k, s_l \in Loc_s,\ s \in Steps\}$, *labeled with copy_trans and*
    $\mu := \{(v, v') \mid v \models x \geq \varepsilon \wedge v' = v[x := 0]\}$ *for some* $0 < \varepsilon << 1$.
- $\forall\ s \in Steps$ *with* $Dyn(s) = ((cond_1 : ODE_1), ..., (cond_m : ODE_m))$, $Loc_s = \{s_1, ..., s_{m+1}\} : Act(s_i) = ODE_i \cup \{x' = 1\}$, $\forall\ i \in \{1, ..., m\}$ *and* $Act(s_{m+1}) = \{x' = 1\}$,
- $\forall\ s \in Steps$ *with* $Dyn(s) = ((cond_1 : ODE_1), ..., (cond_m : ODE_m))$, $Loc_s = \{s_1, ..., s_{m+1}\} : Inv(s_i) = cond_i \wedge (\bigwedge_{j=1}^{i-1} cl(\neg cond_j))$, $\forall\ i \in \{1, ..., m\}$ *and* $Inv(s_{m+1}) = (\bigwedge_{j=1}^{m} cl(\neg cond_j))$,

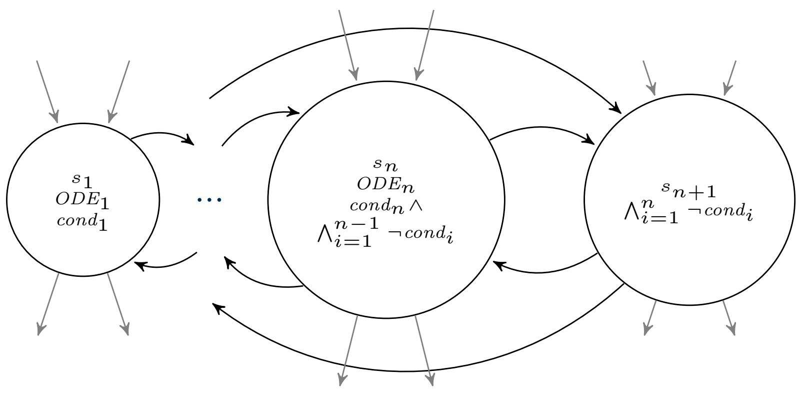

s_1
ODE_1
$cond_1$
$\cdots$
s_n
ODE_n
$cond_n \wedge$
$\bigwedge_{i=1}^{n-1} \neg cond_i$
s_{n+1}
$\bigwedge_{i=1}^{n} \neg cond_i$

Figure 2.11: Splitting a step for each assigned conditional ODE plus one, containing all conditions negated.

- $L := \{action\_sync, read\_sync, copy\_trans\}$,
- $sync : Edge \to L$ *with*
$$L(t) = \begin{cases} action\_sync, & \text{if } t \in Edge_{ActionSync} \cup Loop_{ActionSync} \\ read\_sync, & \text{if } t \in Loop_{ReadSync} \\ copy\_trans, & \text{if } t \in Edge_{CopyTrans} \end{cases},$$
- $Init := \{(s_0, v_0)\}$, *where* v_0 *is the initial variable valuation of* Var_H *and*
- $\sqsubset_H \subseteq Asg \times Asg$ *with for all* $v \in V$ *and for all variables* $b = act_1^{-1}(\mathbb{R}), c = act_2^{-1}(\mathbb{R}) \in Act : h(v, b, act_1) \sqsubset_H h(v, c, act_2)$ *iff* $act_1 \sqsubset act_2$.

The function cl *replaces each* $>$ *with* $\geq$ *and each* $<$ *with* $\leq$

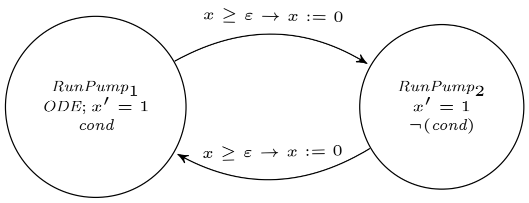

$x \geq \varepsilon \to x := 0$
$RunPump_1$
$ODE; x' = 1$
$cond$
$RunPump_2$
$x' = 1$
$\neg(cond)$
$x \geq \varepsilon \to x := 0$

Figure 2.12: The splitting of step RunPump (Figure 2.5) with $cond := min_1 \wedge \neg max_2 \wedge P_1 \wedge mixer$ and $ODE := h_2' = c_2; h_1' = c_1$.

Chapter 3

Set- and reset-qualifiers

In the IEC standard [Int03] the syntax of SFCs is given. Besides the action qualifiers presented in [NÁ12], the industry standard introduces some more qualifiers. Two of these qualifiers are the *set* (**S**) and the *reset* (**R**) qualifier. In the following, the set of action qualifiers, which is supported by (H)SFCs, will be extended to the *set* and the *reset* qualifiers.
Actions qualified by *set* will be executed repeatedly until a *reset* takes place. That means if an action was set, it will be executed even if the containing step is deactivated, i.e. when this action does not appear in an action block of an active step. In addition to the reset of **S**-qualified actions, the *reset* qualifier allows to prevent the execution of **N-**, **P1-** and **P0**-qualified actions. That means if e.g. an **N**-qualified action is active and then the same action, but **R**-qualified becomes active, the action will not be further executed. If a step contains the same action both *set*- and *reset*-qualified, the reset dominates and the action will not be executed. Furthermore a *set*- or *reset*-qualified action will be executed before all other actions. That means changes, which are made in the *set*-qualified action influence the effect of other actions. In particular, actions which are reset will never be executed as long as a step, which contains the *reset* action, is active, even if the same action occurs **N-**, **P1-**, **P0**-qualified in the set of active actions.
We begin by extending our simple mixer model to *set* and *reset* qualifiers. Afterwards we continue by defining the syntax and semantics of SFCs, which contains *set*- and *reset*-qualified actions and describe in the last part of this chapter the transformation of an (H)SFC into a hybrid automaton (Figure 3.1).
In the following we call the set of (H)SFCs, which contains *set*- and *reset*-qualified actions $(H)SFC_{Set}$. The set of hybrid automata, which contain *set*- and *reset*-qualified actions is denoted with H_{Set}. We introduce an extension of the syntax and semantics of [NÁ12], so that HSFCs are able to handle *(re)set* qualified actions. Furthermore, we give a method to transform this adapted HSFC into a hybrid automaton, which enables verification of *set*- and *reset*-qualified SFCs.

$$\begin{tikzcd}
 & & & & \text{HA} + \text{ODE} \\
{} \arrow[r] & \text{SFC} \arrow[r] \arrow[ddrr, red] & \text{HA} \arrow[r] \arrow[urr] & \text{HA}_{Timed} \arrow[r] \arrow[d] & \text{HA}_{Timed} + \text{ODE} \\
 & & & \text{HA}_{Set+Timed} \arrow[r] & \text{HA}_{Set+Timed} + \text{ODE} \\
 & & & \text{HA}_{Set} \arrow[r, red] \arrow[u] & \text{HA}_{Set} + \text{ODE}
\end{tikzcd}$$

Figure 3.1: Overview of the transformations.

3.1 Simple mixer model - *Set* extension

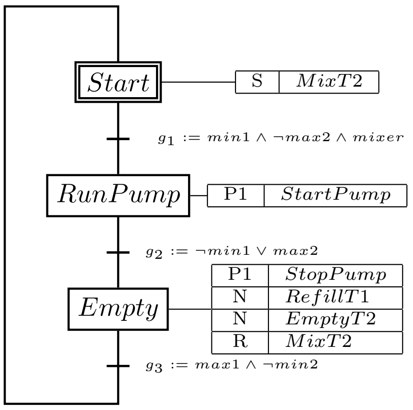


Figure 3.2: Simple mixer model SFC.

We extend our simple mixer model with *set*/*reset* qualifiers, by replacing the actions *StartMixer* and *StopMixer* through one action *MixT2*, which will be set in the first and reset in the last step (Figure 3.2).

3.2 SFC_{Set} syntax

In this first step we extend the SFC syntax given in Definition 1 to *set* and *reset* qualifiers. Therefore we adapt the definitions and descriptions of Section 2.2.1. From now on the set action qualifier will be abbreviated with **S** and the reset action qualifier with **R**.
An SFC $C \in SFC_{Set}$ is an SFC whose action qualifiers are chosen from the set $\{N, P0, P1, \mathbf{S}, \mathbf{R}\}$.

Definition 11 (SFC_{Set}) *An SFC* $C = (\mathit{Var}, \mathit{Steps}, \mathit{Act}, s_0, \mathit{Trans}, \mathit{Blocks}, \sqsubset, \prec, \mathit{Hist}) \in SFC_{Set}$ *with* set *and* reset *qualifiers consists of*

- *Var, Steps, Act,* s_0*, Trans,* $\sqsubset$*,* $\prec$*, Hist as in Definition 1 and*
- $\mathit{Blocks} : \mathit{Steps} \to 2^{B_{Act}}$ *with*
$$B_{Act} := \{(q, a) \mid a \in \mathit{Act},\ q \in \{P1, N, P0, \boldsymbol{S}, \boldsymbol{R}\}\}$$

3.3 SFC_{Set} semantics

We regarded the syntax of the set and reset action qualifiers in Section 3.2. Now we define the corresponding semantics. Therefore we extend the semantics of SFCs introduced in Section 2.2.2.
The main variation to Section 2.2.2 depends on the computation of active actions. In Section 2.2.2, we computed the set of active actions by considering the currently active steps. For an active step, we check each of its action blocks and add the corresponding action to the set of active actions in dependency to its action qualifier. That means, e.g., a **P1**-qualified action will be added to the sequence of active actions if the corresponding step becomes active and a **P0**-qualified action, if it becomes inactive. We will now consider the two new

action qualifiers *set* and *reset*. With *set* it is possible to add an action to the set of active actions independently from its step activity. That means, an action once set in an active step will be executed in the following steps and cycles until it will be reset. In addition to the reset of set actions the *reset* qualifier can also prevent the execution of **P1-**, **N-** and **P0-**qualified actions
We can conclude: The set of active actions, which we have computed in Section 2.2.2 will now extended to *set*-qualified actions, which are not reset and reduced by those **P1-**, **N-** and **P0-** qualified actions, which were reset.

Let us now introduce the SFC$_{Set}$ configuration tuple $(\sigma, readyS\ , activeS\ , activeA\ , storedA\ ) \in \Sigma \times 2^{\overline{Steps}} \times 2^{\overline{Steps}} \times 2^{\overline{Act}} \times 2^{\overline{Act}}$ to handle *set* and *reset*-qualified actions. Let $Conf^{Set}$ be the set of all configuration tuples. We add a set *storedA*, which contains *set* actions. An action will be added to the set of stored actions *storedA*, if it occurs *set*-qualified in an active step and will be removed from it, if it occurs *reset*-qualified. Thereby the *reset* qualifier dominates.
We need this set of stored actions *storedA* to know in the following steps, in which the *set* actions not explicitly appear, that they are set and thus they must be executed.
The set of set actions must be added to the computed sequence of active actions. We use the function $computeActiveSets_{Set}$ to compute $activeA'$ (Algorithm 2).

Definition 12 (Transition relation) *The transition relation* $\rightarrow \subseteq Conf^{Set} \times Conf^{Set}$ *is defined as a configuration change. For a transition* $(c, c') \in \rightarrow$ *with* $c := (\sigma, readyS\ , activeS\ , activeA\ , storedA\ )$ *and*
$(\sigma', readyS', activeS', activeA', storedA') =: c'$*, we define*

- $readyS' = (readyS\ \setminus source(taken(C, c))) \cup target(taken(C, c))$.
- $(activeS', unsortedActiveA, storedA') =$
  $computeActiveSets_{Set}(readyS', \emptyset, storedA\ , C, c, activeA\ \cap \overline{C}, storedA\ , \emptyset)$
- $activeA' = sort(unsortedActiveA', \sqsubset)$.
- $\sigma' = f(\sigma) = (a_m \circ ... \circ a_1)(\sigma)$ *where* $activeA' = a_m \circ ... \circ a_1$ *and* $f \in F$.

The function $computeActiveSets_{Set}$ *is defined in Algorithm 2 and sort arranges the actions descending with respect to the given order* $\sqsubset$ *as a sequence.*

Algorithm 2 is an extension of Algorithm 1. They are distinguishable in the additional function of Algorithm 2, which allows to consider *(re)set*-qualified actions.
Therefore we add two additional cases to the if-branch in line 8. If an active step contains a *set*-qualified action it must be added to the sequence of active actions $activeA'$ (if it is not already contained) and to the set actions $storedA'$. But if an active step contains a *reset*-qualified action, it must be removed from the set of active actions $activeA'$ and the set of stored actions $storedA'$.
Because of the dominance of *reset*-qualified actions, we must consider the case that a *reset* arrives before a *set*. Therefore we introduce a set *resetA*, which stores all *reset*-qualified action. We check if an action is member of this set, before we add it to the sequence of active actions $activeA'$ or to the *set* actions $storedA'$.
To handle the case that an action will be set in a nested SFC, we add *resetA* to parameters of the recursive function call in line 17. This allows us to compute

Algorithm 2: $computeActiveSets_{Set}($ $readyS', activeS, activeA, C, c, activeSFCs, storedA, resetA)$

```
input  : $readyS', activeS, activeA,$
         $C = (Var, Steps, Act, s_0, Trans,$
         $Blocks, \sqsubset, \prec, Hist), c, activeSFCs, storedA, resetA$
output: $activeS', activeA', storedA'$
/* Add the local active steps of $C$ to $activeS'$.                */
1  if $Hist = 1 \lor C \in activeSFCs$ then
2      $activeS' := activeS \cup (Steps \cap readyS')$;
3  else
4      $activeS' := activeS \cup \{s_0\}$;
/* Add the former setted actions to the set of active
   actions.                                                        */
5  $activeA' := activeA$;
/* Stores the former set actions.                                  */
/* Added:                                                          */
6  $storedA' := storedA$;
/* Collect the local active actions and their qualifiers.          */
7  foreach $s \in Steps, b = (q, a) \in Blocks(s)$ do
8      if $([q = exit \land s \in source(taken(C, c))) \lor (q = entry \land s \in$
       $target(taken(C, c))) \lor (q = do \land s \in activeS'] \land a \notin activeA' \land a \notin resetA)$
       then
9          $activeA' := activeA' \circ a$;
       /* Add set actions, which are not resetted.                 */
       /* Added:                                                   */
10     else if $(q = set \land s \in activeS' \land a \notin resetA)$ then
11         $storedA' := storedA' \cup \{a\}$;
12     else if $(q = reset \land s \in activeS')$ then
13         $resetA := resetA \cup \{a\}$;
           $storedA' := storedA' \backslash \{a\}$;
           $activeA' := activeA' \backslash a$
14 end
/* Compute $activeA'$ and $activeS'$ for each active nested SFC    */
15 foreach $s \in Steps \cap activeS', b = (q, a) \in Block(s)$ do
16     if $a \in \overline{C}$ then
17         $(activeS', activeA', storedA') :=$
           $\text{computeActiveSets}_{\text{Set}}(readyS', activeS', activeA',$
                     $a, c, activeSFCs, storedA', resetA)$;
18 end
/* Add active stored actions to the active actions.                */
/* Added:                                                          */
19 foreach $a \in storedA' \land a \notin activeA'$ do
20     $activeA' := activeA' \circ a$;
21 end
22 return $(activeS', activeA', storedA')$;
```

the set of active actions including the *set* actions for an SFC and its nested child SFCs.
Because a *reset* affects *set*-qualified actions as well as **N**-, **P1**- and **P0**-qualified actions, we must adapt the branch in line 8 to check, whether an action is already reset. Following this reason we also remove a reset action from the sequence of active actions (line 13). So a *reset*-qualified action is absolutely dominant and able to prevent any other qualified actions from being executed.
Let us consider the case that an action appears **N**-qualified, while it was already *set* in a step before. In that case we will only execute the action once. The algorithm checks whether an action is already member of the execution sequence before it will be added.
Algorithm 2 returns an unsorted sequence of active actions. Therefore we apply the *sort(...)* function on it, which arranges the sequence descending with respect to $\sqsubset$.

3.3.1 Additional semantic definitions for SFC$_{Set}$

We extend Definition 4, which provides functions to get **N**-, **P0**- and **P1**-qualified actions of an SFC, with six further definitions. As in Definition 4 we define functions to get the sets of *set*- and *reset*-qualified actions, both nested and for a local step.

Definition 13 (Specific action set function) *Let* $C = (Var, Steps, Act, s_0, Trans, Blocks, \sqsubset, \prec, Hist) \in SFC_{Set}$ *an SFC and* $c = (\sigma, readyS\ , activeS\ , activeA\ , storedA\ ) \in Conf^{Set}$ *a configuration of* C*. We define*

- $Set : Steps \to 2^{Act}$ *with* $Set(s) = \{a \mid (S, a) \in Blocks(s)\}$
- $Reset : Steps \to 2^{Act}$ *with* $Reset(s) = \{a \mid (R, a) \in Blocks(s)\}$
- $Setter : Steps \to 2^{Act}$ *with* $Setter(s) = Set(s) \cup Reset(s)$

We extend our definition to nested SFCs as follows:

- $Set_{Nested} : Conf^{Set} \to 2^{Act}$ *with* $Set_{Nested}(c) = \bigcup_{s \in activeS}\ Set(s)$
- $Reset_{Nested} : Conf^{Set} \to 2^{Act}$ *with* $Reset_{Nested}(c) = \bigcup_{s \in activeS}\ Reset(s)$
- $Setter_{Nested} : Conf^{Set} \to 2^{Act}$ *with* $Setter_{Nested}(c) = \bigcup_{s \in activeS}\ Setter(s)$

We will now explain the transformation process of a SFC$_{Set}$ and come to its formal definition in the following.

3.4 $HSFC_{Set} \to$ Hybrid automaton

Now we give a transformation from HSFC$_{Set}$ to hybrid automata. In contrast to HSFCs, there exist several approaches to analyses and thus verify hybrid automata [ACH$^+$95].
We begin by defining a method for the transformation of SFCs to hybrid automata and extend it to HSFCs in the following. The transformation is based on the transformation of an SFC, which is described in Section 2.5.1.
We extend the variable set Var_H of the automaton, which is the result of an SFC transformation (Definition 9), with *setter* variables Var_{Set}. That means we

add one variable for each action, that can be *set*. These variables can be seen as boolean flags. The *setter* variables of *set*-qualified actions are per default initialized to 0.

Definition 14 (*Setter* variables) *The set of setter variables is defined as*

$$Var_{Set} := \{\xi_a \mid (q, a) \in Set(s),\ s \in Steps\}$$

An appearing **S** qualifier will assign the value 1 to the corresponding setter variable, while an appearing **R** qualifier will assign the value 0 to it (Definition 15). Since an action will only be executed if the corresponding setter variable is true, we control the execution of actions.
A *set*- or *reset*-qualified action in an SFC step influences all other actions of the step. That means if, e.g., a new value was assigned to a variable x by a *set*-qualified action the following actions of the step must use this adapted value of the variable x in their computations. Therefore *set* and *reset* actions are higher priorized and executed before other actions. Within these *(re)set* actions it holds that *set* actions are executed earlier than *reset* actions. This behavior is defined in [BHLE04].
We define an action a_ξ^{Set}, which valuates a setter variable ξ to true and an action a_ξ^{Reset}, which valuates it to false. These actions will be executed first, before all other actions. Thereby a *set* action must be executed before a reset action, since the *reset* action dominates and the setter variable valuation will not further change after the *reset* was executed. The priorization within the setter actions will be adopted from the priorization of their corresponding *set* actions.

Definition 15 (Set/ Reset Control Action) *For each setter variable* $\xi \in Var_{Set}$ *we define actions* a_ξ^{Set}, a_ξ^{Reset} *with*

- $a_\xi^{Set} : \xi := 1$
- $a_\xi^{Reset} : \xi := 0.$

We write a_ξ° *if an action sets or resets a variable and define the following action priority. Let* $C = (Var, Steps, Act, s_0, Trans, Blocks, \sqsubset, \prec, Hist)$ *be an SFC. We extend the order* $\sqsubset$ *and the set of actions Act to* $Act \cup \{a_\xi^{Set}, a_\xi^{Reset} \mid \xi \in Var_{Set}\}$ *as follows:*

- $\forall \xi \in Var_{Set}\ \forall a \in Act : a_\xi^\circ \sqsubset a$
- $\forall \xi_{a_i},\ \xi_{a_j} \in Var_{Set} : a_i \sqsubset a_j \Rightarrow a_{\xi_{a_i}}^\circ \sqsubset a_{\xi_{a_j}}^\circ$
- $\forall \xi_i,\ \xi_j \in Var_{Set} : a_{\xi_i}^{Set} \sqsubset a_{\xi_j}^{Reset}$

Due to the new introduced variables, we now need to introduce adapted transition guards. Our approach bases on the idea of [NÁ12] to decode the action blocks of an SFC into transitions of the hybrid automata (Section 2.5.1).
The aim is to execute those actions where the corresponding setter variable is valuated to true. Because there are different possible combinations of true-valuated setter variables, we provide for each of these combinations one transition, which executes the corresponding actions. That means we extend the guard of the transitions (computed in Section 2.5.1) by setting the true-valuated setter

variables and the negated false-valuated variables in conjunction to the already existing guards (Figure 3.3). We do this for every existing *action_sync* labeled transition and for each possible evaluation of the setter variables. Finally we have exactly one transition for each combination of setter variables and for each existing transition, which executes the corresponding actions.

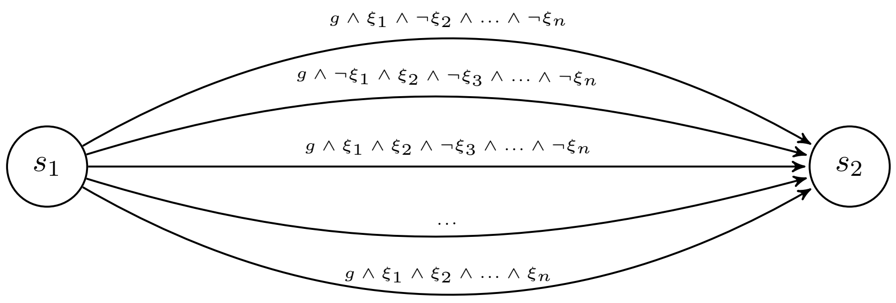

$g \wedge \xi_1 \wedge \neg\xi_2 \wedge \ldots \wedge \neg\xi_n$
$g \wedge \neg\xi_1 \wedge \xi_2 \wedge \neg\xi_3 \wedge \ldots \wedge \neg\xi_n$
s_1
$g \wedge \xi_1 \wedge \xi_2 \wedge \neg\xi_3 \wedge \ldots \wedge \neg\xi_n$
s_2
$\ldots$
$g \wedge \xi_1 \wedge \xi_2 \wedge \ldots \wedge \xi_n$

Figure 3.3: All combinations of true-valuated setter variables be added.

As future work we could apply a reachability analysis on a given SFC, before the transformation starts. In SFCs it is possible that some steps will never be reached with a dedicated set of true-valuated setter variables. We could also check, whether an action will be reset. If this is the case, we can omit its setter variable.

In the following we provide an algorithm, which computes all possible setter transitions. We begin by describing the idea of the algorithm in an informal way. After that we give a formal definition, which enables the implementation.
Because of efficiency considerations, we need an algorithm, which computes each combination only once and aborts if all combinations are found. The idea is to iterate over the amount of possible combinations. We consider a given iteration in its binary representation. Each place of the binary expression represents one setter variable (Figure 3.4) and we take those setter variables, which have ones in the corresponding places of the boolean expression. Because every number has a unique binary representation, we get for each number a unique set of setter variables, which we can set to true. In particular we get each combination exactly once.

$$\begin{array}{rl} Binary\ number: & \square \cdots \square\square\square\square\square\square\square \\ Variables: & \xi_n \cdots \xi_7\,\xi_6\,\xi_5\,\xi_4\,\xi_3\,\xi_2\,\xi_1 \\ & \text{MSB} \qquad\qquad\qquad\quad \text{LSB} \end{array}$$

Figure 3.4: Each place represents one setter variable.

Let us assume that we have n setter variables and we want to compute all possible combinations. We can set each of these setter variables true or false. That means in our binary representation: We can set the corresponding places either to one or to zero.

$$\#\ combinations\ :\ \underbrace{2 * \ldots * 2}_{n} = 2^n \tag{3.1}$$

So we can choose n times between two different states (zero and one) and get the number of possible combinations 2^n. We conclude: We must count from 0 to $2^n - 1$ to get all possible combinations of n different variables.

We are now able to iterate over all possible combinations, but need a function, which concatenates the setter variables, respectively. Therefore we define the *guard* function:

Algorithm 3: guard(n, *combination*).

```
   input  : $n$, $combination$
   output: $setterVars$
   /* Initialize the output.                                  */
1  $setterVars := \emptyset$;
   /* For each place in the number:                           */
2  for $pos = 1$ to $n$ do
       /* Check for the first place if it is one.             */
3      if $1$ & $combination$ then
4          $setterVars := setterVars \cup \{\xi_{pos}\}$;
       /* Shift the number and increment the position.        */
5      $combination := combination >> 1$;
6  end
7  return ($setterVars$);
```

The algorithm extracts the different combinations of setter variables from a number in binary representation. Thereby it shifts a number with n places n times right, where left is the most significant bit (MSB)[1]. In every step we take the conjunction of the shifted number with one and check in this way, whether the first place contains a one. If this is the case, we add the corresponding variable to the set of setter variables $setterVars$.
To get all possible combinations we must execute the algorithm for every number from 0 to $2^n - 1$, where n is the number of setter variables.

We will now take a look on the efficiency of the algorithm. For n setter variables we must execute the algorithm 2^n times. For each number we do n right shifts so we get a time complexity of $O(n2^n)$. We regard the practical usability of the transformation, in particular of this algorithm, in Section 7.

To keep it intuitive we begin with the definition of the edge set and give the definition of the whole transformation in the following. Thereby we collect all actions assigned to a step and add transitions for each combination of activated actions to the step. Every transition will only execute a specific set of actions and the taken transition depends on the different possibilities to *reset* some actions. Through a *reset* we can prevent that an action will be executed. The *reset* remains preventing as long as the corresponding step is active. So e.g. if an action occurs **N**- and **R**-qualified in a step, it will not be executed.

Example 2 (Guard function) Let $\{\xi_1, \xi_2, \xi_3\}$ be a set of $n = 3$ setter variables. Their exists $2^3 = 8$ possible combinations and we want to compute

[1]We abbreviate least significant bit with LSB.

the fifth combination (*combination* $= 5$) $guard(3,5)$:

- $pos = 1\ :\ 1\ 0\ 1\ \&\ 1 \Rightarrow setterVars := setterVars \cup \{\xi_1\}$
  $1\ 0\ 1\ >>\ 1 = 0\ 1\ 0,\ pos = pos + 1$

- $pos = 2\ :\ 0\ 1\ 0\ \&\ 1$
  $0\ 1\ 0\ >>\ 1 = 0\ 0\ 1,\ pos = pos + 1$

- $pos = 3\ :\ 0\ 0\ 1\ \&\ 1 \Rightarrow setterVars := setterVars \cup \{\xi_3\}$
  $0\ 0\ 1\ >>\ 1 = 0\ 0\ 0,\ pos = pos + 1$

We computed the set: $setterVars = \{\xi_1, \xi_3\}$.
Applying the algorithm we extract a set of setter variables from a given number (Example 2). We execute the Algorithm 3 for each combination of setter variables (Example 3) and add those setter variables negated, which are not part of the combination.

Example 3 (Guard function) Let $Var_{Set} := \{\xi_1, \xi_2, \xi_3\}$ be the *setter* variable set with $|Var_{Set}| = p = 3$. We compute:
$\bigcup_{i=0}^{2^3-1}\{p,i)\} = \bigcup_{i=1}^{7}\{guard(p,i)\}$

- $i = 0 : guard(3,0) :$
  $0 =\ 0\ 0\ 0 \rightarrow \{\neg\xi_1, \neg\xi_2, \neg\xi_3\}$
- $i = 1 : guard(3,1) :$
  $0 =\ 0\ 0\ 1 \rightarrow \{\neg\xi_1, \neg\xi_2, \xi_3\}$
- $i = 2 : guard(3,2) :$
  $0 =\ 0\ 1\ 0 \rightarrow \{\neg\xi_1, \xi_2, \neg\xi_3\}$
- $i = 3 : guard(3,3) :$
  $0 =\ 0\ 1\ 1 \rightarrow \{\neg\xi_1, \xi_2, \xi_3\}$
- $i = 4 : guard(3,4) :$
  $0 =\ 1\ 0\ 0 \rightarrow \{\xi_1, \neg\xi_2, \neg\xi_3\}$
- $i = 5 : guard(3,5) :$
  $0 =\ 1\ 0\ 1 \rightarrow \{\xi_1, \neg\xi_2, \xi_3\}$
- $i = 6 : guard(3,6) :$
  $0 =\ 1\ 1\ 0 \rightarrow \{\xi_1, \xi_2, \neg\xi_3\}$
- $i = 7 : guard(3,7) :$
  $0 =\ 1\ 1\ 1 \rightarrow \{\xi_1, \xi_2, \xi_3\}$

We will now introduce three functions for notational convenience.

Definition 16 (*conj*- function) *Let Var_{Set} be the set of all setter variables and $set \subseteq Var_{Set}$. We define the conjunction function $conj : 2^{Var_{Set}} \rightarrow G^{Var_{Set}}$ as*

$$conj(set) := \bigwedge_{\xi \in set} \xi \ \wedge\ \bigwedge_{\xi' \in Var_{Set} \setminus set} \neg\, \xi'.$$

The *conj*- function will give the conjunction of all variables ξ in the set *set* and the negated variables $\neg\xi$, which are not member of the set $Var_{Set} \setminus set$.
We will define the second function, which returns the corresponding actions for a set of setter variables:

Definition 17 (*acts*- function) *Let Var_{Set} be the set of all setter variables and $set \subseteq Var_{Set}$. We define the function $acts : 2^{Var_{Set}} \to 2^{Asg}$ as*

$$acts(set) := \{a \mid \xi_a \in set\}.$$

Since the *acts* function returns a set of actions we define a *sort(...)* function, which arranges a set as a descending sequence with respect to a given order $\sqsubset$.

Definition 18 (*sort*- function) *For a set $set = \{a_1, ..., a_n\}$ of assignments with $a_1 \sqsubset ... \sqsubset a_n$. We define the function $sort : 2^{Asg} \times 2^{\sqsubset} \to Asg^*$ as*

$$sort(set, \sqsubset) := a_n \circ ... \circ a_1.$$

Before we finally introduce the whole transformation from an SFC$_{Set}$ into a hybrid automaton, we show how we can create its transitions. This is one of the main parts of the transformation and moreover the most complex.
The definition of the *Edge* set (Definition 19) follows two main ideas. First the adaption of the transition guard in dependency to the current set of setter variables and secondly the controlling of *setter* variables by executing *(re)setter* control actions (Definition 15).
We add for each combination of *setter* variables and each transition one further transition. The combinations, which we had computed using the *guard* function, determines the assignments of actions to the added transitions. So we will add those actions, where the corresponding *setter* variable was set in the guard. We do this for transitions from one location to another as well as for variables, the corresponding actions will be executed.
The valuation of a setter variable will be influenced by incoming transitions, where *(re)setter* actions are assigned in relation to the *set*/ *reset*-qualified actions in the SFC. That means a *(re)set* action in an SFC will be (de)activated by adding (re)setter actions to the incoming transitions of the corresponding location in the hybrid automaton. Since actions, which are not set, must be reset in each cycle anew, we add *(re)set* control actions for **N**- and **P0**-qualified actions to the self loop. The corresponding action will be assumed as set in each cycle and a later executed resetter action can prevent its execution. *Reset* control actions are only added to a self loop to prevent the execution of actions, which occurs *reset*-qualified in the same step. So we handle the case that an action occurs **N** or **P0** and **R**-qualified in the same set of action blocks, which is assigned to a step. Because we exclude parallel branches and nested SFCs, we can only reset a **N**, **P1** or **P0**-qualified action by adding the action *reset*-qualified to the same step. But as a future work, we can assume that an action will be in a nested SFC or a parallel branch reset. Then the action must be again executed after the *reset* action becomes inactive.
To enable true-valuated *setter* variables in the initial state of the SFC, we must adapt the initial valuation of setter variables in the hybrid automaton. So the corresponding setter variable of an action, which was set but not reset in the initial state, will be valuated to true in the beginning.

Definition 19 (Edge set - *Edge*$_{Set}$) *Let SFC$_{Set}$ $C = (Var, Steps, Act, s_0, Trans, Blocks, \sqsubset, \prec, Hist) \in SFC_{Set}$ be an SFC without parallel branches and nested SFCs. Let $s \in Steps$ be a step with outgoing transitions $t_1, ..., t_q \in Trans$, $t_j = (s, g_j, s_j)$ for $j = 1, ..., q$. Let $t_1 \prec ... \prec t_q$ be the transition priorities. We define $Edge_{set} := Edge_{ActionSync} \cup Loop_{ActionSync} \cup Loop_{ReadSync}$, with*

- $Edge_{ActionSync} := \{(s, \mu_{i,j}, s_j) \mid j \in \{1, ..., q\}, i \in \{1, ..., 2^p\}\}$, *labeled with action_sync and*
  $\mu_{i,j} := \{(v, v') \mid v \models g_j \ \wedge \ \bigwedge_{k=1}^{j-1} \neg g_k \wedge conj(guard(p, i)) \ \wedge$
  $sort(acts(guard(p, i)) \cup ((Do(s_j) \cup Entry(s_j) \cup Exit(s)) \backslash Reset(s_j)), \sqsubset) \circ sort(Setter_{control_1}, \sqsubset)(v) = v'\}$, *with*

  $Setter_{control_1} = \{a^{Set}_{\xi_{a'}} \mid a' \in Set(s_j)\} \cup \{a^{Reset}_{\xi_{a'}} \mid a' \in Reset(s_j)\}$ *and* $p = |\bigcup_{s' \in Steps} Set(s')|$,

- $Loop_{ActionSync} := \{(s, \mu_i, s) \mid i \in \{1, ..., 2^p\}\}$, *labeled with action_sync and*
  $\mu_i := \{(v, v') \mid v \models \bigwedge_{k=1}^{q} \neg g_k \ \wedge \ conj(guard(p, i)) \ \wedge$
  $sort(acts(guard(p, i)) \cup (Do(s) \backslash Reset(s)), \sqsubset) \circ sort(Setter_{control_2}, \sqsubset)(v) = v'\}$, *with*

  $Setter_{control_2} = \{a^{Set}_{\xi_{a'}} \mid a' \in Set(s)\} \cup \{a^{Reset}_{\xi_{a'}} \mid a' \in Reset(s)\}$ *and* $p = |\bigcup_{s' \in Steps} Set(s')|$,

- $Loop_{ReadSync} := \{(s, \mu, s) \mid s \in Steps\}$, *labeled with read_sync and* $\mu := \{(v, v') \mid v\,|_{Var} = v'\,|_{Var} \wedge \forall\, a \in Var: \ v'(a) = v'(a_{global})\}$.

The functions guard(...), conj(...), acts(...), sort(...) are defined in Algorithm 3, Definition 16, Definition 17, Definition 18.

Now we can define the transformation from an SFC$_{Set}$ to a hybrid automaton as follows. Thereby we extend Definition 9, which describes the transformation of an SFC.

Definition 20 (SFC$_{Set}$ $\rightarrow$ Hybrid automaton) *Let $C = (Var, Steps, Act, s_0, Trans, Blocks, \sqsubset, \prec, Hist)$ be an SFC without parallel transitions and without nested SFCs and $H = (Loc, Var_H, Edge, Act_H, Inv, L, sync, Init, \sqsubset_H)$ its transformation into a hybrid automaton with*

- $Var_H = Var \cup Var_{Set}$
- $Edge = Edge_{Set}$ *with* $Edge_{Set}$ *as in Definition 19.*
- *Loc,* Act_H*, Inv, L, sync, Init and* $\sqsubset_H$ *are defined similarly as in Definition 9.*

Our simple mixer model with only one setter action and two **N**-qualified actions can be computed by adding all possible combinations of set actions to the locations (Figure 3.5). So e.g. let us consider the location *Start*. If we transform *Start* into a hybrid automaton without consideration of set actions, we get only one self loop, which executes the assigned **N** action (Figure 2.10). But if we consider set actions, we must duplicate the number of self loops, because we need one, which will be taken if the action is not set and one, if it is set.

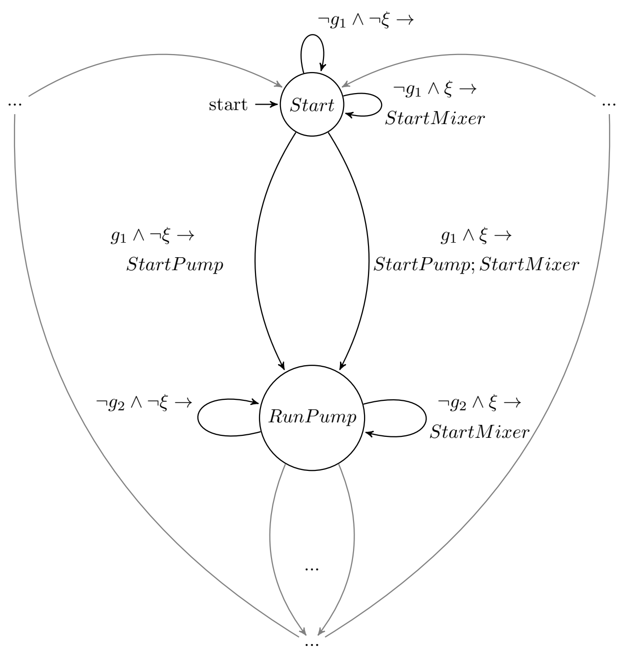

$\neg g_1 \wedge \neg\xi \rightarrow$
...
start
Start
$\neg g_1 \wedge \xi \rightarrow$
StartMixer
...
$g_1 \wedge \neg\xi \rightarrow$
StartPump
$g_1 \wedge \xi \rightarrow$
StartPump; *StartMixer*
$\neg g_2 \wedge \neg\xi \rightarrow$
RunPump
$\neg g_2 \wedge \xi \rightarrow$
StartMixer
...
...

Figure 3.5: Simple Mixer Model (restricted to two steps) - Hybrid automaton.

We have shown how to transform an SFC with *(re)set*-qualified actions into a hybrid automaton. Let us now consider how to transform an HSFC with *(re)set*-qualified actions into a hybrid automaton.
Definition 10 describes how to transform a common HSFC into a hybrid automaton. It begins by transforming the corresponding SFC into a hybrid automaton using Definition 9.
Definition 10 adds conditional ODEs to a hybrid automaton, which was created before by Definition 9. Because we do not change anything on the conditional ODE semantics and syntax, we are able to apply the Definition 10 on our *(re)set*-qualified SFC transformation Definition 20.
In this way we can transform an HSFC$_{Set}$ with *(re)set*-qualified actions into a hybrid automaton.

3.4.1 Complexity consideration

In the following we will give an overview about the complexity of transformations. Thereby we consider the amount of locations, variables and transitions, which will be created in the different transformations. We begin with the analysis of the transformation of an SFC into a hybrid automaton and contrast it to the transformation of a HSFC into a hybrid automaton. Based on this results we will analyses the transformation of an SFC, which contains *setter* actions.

SFC $\rightarrow$ HA

Let C be an SFC with n steps, m transitions and k variables.
We create for each step one location in the hybrid automaton, so we have n

locations. We do not change the set of variables. Therefore we get k variables in the hybrid automaton. For each transition in the SFC we create a transition in the hybrid automaton and additionally we provide each step with a self loop. We further add one synchronization transition per step for the synchronization automaton. So we get $m + 2n$ transitions in the hybrid automaton.

HSFC → HA

Let *HSFC* be an HSFC with n steps, m transitions, $k + k'$ variables and p conditional ODEs. k' represents the number of continuous variables. Let p_s be the number of ODEs mapped to a step s.
We can now compute the number of locations, which will be created by the transformation as follows:

$$\sum_{s \in Steps} (p_s + 1) \leq n * (p + 1) \tag{3.2}$$

We can estimate that $p_s = p$ in the worst case. This is the case if all conditional ODEs are mapped to every step. So we can estimate the number of locations as:

$$O(np) \tag{3.3}$$

If a step contains no conditional ODEs we count it as one, what means the unaltered step itself. If it contains conditional ODEs we add one, to count the step, which will be taken if no condition holds.
The set of variable will be extended with one variable, which is needed to prohibit zeno behavior, so that we get $k + k' + 1$ variables in the resulting hybrid automaton. The number of transitions can be computed as follows:

$$2 * \sum_{s \in Steps} (p_s + 1) + \sum_{s \in Steps} out(s) * (p_s + 1) + \sum_{s \in Steps} \sum_{i=1}^{p_s} 2 * (p_s - i) \tag{3.4}$$

The first sum represents the number of self loops (*action_sync* and *read_sync*), which must be added to the locations, while the second counts the transitions, which are also part of the SFC. $out(s)$ returns the number of outgoing transitions of step s. The last sum counts the transitions, which will be added between splitted locations. These are the locations, where the corresponding step in the HSFC contains a conditional ODE system.
We transform the last sum into a closed formula and with the assumption above we get

$$\sum_{s \in Steps} (p^2 + p + p * out(s) + out(s) + 2). \tag{3.5}$$

$\sum_{s \in Steps} out(s) = m$, because the sum of all outgoing transition is equal to the number of all existing transitions. So we can estimate the number of transitions needed for the transformation with:

$$O(np^2 + pn + pm + m + 2n) \tag{3.6}$$

setter SFC $\rightarrow$ HA

Let C be an SFC with n steps, m transitions and k variables. u represents the number of *set*-qualified actions.
Our approach does not add any further locations to the hybrid automaton and bases on the transformation described in Definition 9. So we get n locations in our resulting hybrid automaton.
Because we create for each *set*-qualified action one *setter* variable, which represents its activation, the resulting hybrid automaton contains $k + u$ variables.
We decode the number of *set* and *reset* qualifiers in the transitions of the automata. Therefore we add for each transition all possible combinations. The idea for the complexity analysis is that we select l variables from a set of $k + u$ variables. These variables are indistinguishable and we cannot select one variable twice. We get the following equation:

$$\sum_{i=0}^{u} \frac{u!}{(u-i)!i!} = 2^u \tag{3.7}$$

We do this for each transition in the SFC and the self loops, which we create in the hybrid automaton. So we get the following number of transitions:

$$(n + m) * 2^u + n \tag{3.8}$$

The additional summand n describes the *read_sync* transitions, which are not adapted by the transformation. In our approach we can transform a *setter* HSFC into a hybrid automaton by transforming first its SFC part into a hybrid automaton. We apply the HSFC transformation thereafter. Therefore we can compute the amount of needed space for the transformation of a HSFC$_{Set}$ by applying the equations given above.

Analyse results

We give an overview of the space complexity in the following table:

	Steps/ Locs.	Variables	Transitions	Set. act.
(H)SFC	n	$k + k'$	m	u
SFC $\rightarrow$ HA	n	k	$m + 2n$	u
set. SFC $\rightarrow$ HA	n	$k + u$	$(m + n) * 2^u + n$	u
HSFC $\rightarrow$ HA	$O(np)$	$k + k' + 1$	$O(np^2 + pn + pm + m + 2n)$	u
set. HSFC $\rightarrow$ HA	$O(np)$	$k + k' + 1$	$O(np^2 + pn + p2^u + 2^u + 2n)$	u

We see that if we use *setter* variables, the number of needed transitions increases exponentially. Therefore optimizations on the transformed automaton can be useful. We give some ideas in the conclusion (Section 7).

CHAPTER 4

Timed action qualifier

Additionally to the qualifiers regarded in Chapter 2 and to the *(re)set* qualifiers presented in Chapter 3, we will now introduce *timed action qualifiers*. They allow to define a delay or an activation period, after or during the action will be executed. We can further add a *set* attribute to those timed action qualifiers, which allow an activation of actions independently from the step activity or the delayed *set* of an action.
We extend the syntax and semantics of SFCs and give instructions to transform a *timed SFC* (SFC_{Timed}), that is an SFC, which contains time-qualified actions, into a hybrid automaton. An SFC, which contains both (re)set- and time-qualified actions will be denoted with $\mathrm{SFC}_{Set+Timed}$. Our approach is based on the hybrid automaton that we get by applying the SFC transformation of Chapter 2 or the SFC_{Set} transformation of Chapter 3, if it contains *setter* variables (Figure 4.1).

$$\begin{tikzcd}
 & & & & \mathrm{HA} + \mathrm{ODE} \\
{} \arrow[r] & \mathrm{SFC} \arrow[r] \arrow[ddrr] & \mathrm{HA} \arrow[r, red] \arrow[urr] & \mathrm{HA}_{Timed} \arrow[r, red] \arrow[d, red] & \mathrm{HA}_{Timed} + \mathrm{ODE} \\
 & & & \mathrm{HA}_{Set+Timed} \arrow[r, red] & \mathrm{HA}_{Set+Timed} + \mathrm{ODE} \\
 & & & \mathrm{HA}_{Set} \arrow[r] \arrow[u, red] & \mathrm{HA}_{Set} + \mathrm{ODE}
\end{tikzcd}$$

Figure 4.1: Overview transformations - Timed.

We give instructions to transform a hybrid automaton (HA or HA_{Set}) into a hybrid automaton, which is able to model timed actions HA_{Timed} and show in the following how to extend it to an automaton, which considers the conditional ODE systems ($\mathrm{HA}_{Timed} + \mathit{ODE}$).
We begin by introducing the different timed action qualifiers and explain their characteristic in detail. Thereafter we come to their syntax and semantics and give the transformation instructions.

4.1 Timed action qualifier characteristics

The difference between the qualifiers of the foregoing chapters and timed action qualifiers is an additional time condition, which has an effect on the time point of the execution of the action.
We distinguish five different timed action qualifiers: *Limited* (**L**), *Delayed* (**D**), *Delayed Set* (**DS**), *Set Delayed* (**SD**) and *Set Limited* (**SL**).
A variable d of type *TIME* is attached to each of them. This will determine the activation time or a delay. For this account, we extend the definition of action blocks (Section 2.2.1) following [BHLE04]. Thereby we convert the variable

d into a real-valued variable. A variable of type *TIME* is introduced by a keyword *TIME#* or its abbreviation *T#*. The keyword can be written both in lower case or in capital letters. We can further specify five different time spaces separately in one variable of type *TIME*. These are *days* (d), *hours* (h), *minutes* (m), *seconds* (s) and *milliseconds* (ms). To specify a time, we can write them, with a leading number, directly one after another or separated by underlines (Example 4). Thereby the specification of the time values represents a descending sequence, that means the highest time type will be placed left and the next highest thereafter and so on. The time values can flood, what means that it is also possible to specify *25 hours* without changing the number of *days*.

Example 4 (Variable of type *TIME*) Let us assume we want to specify 25 days, 6.3 hours, 5 minutes, 1 second and 30 milliseconds. We can write

- $time\#25d6.3h5m1s30ms$, or
- $T\#25d\_6.3h\_5m\_1s\_30ms$.

We can now remove the minutes and write the data type as follows:

- $t\#25d6.3h1s30ms$, or
- $TIMED\#25d\_6.3h\_1s\_30ms$.

This will not be a good format for doing computations on them. Therefore we assume only one value, which is equal to the time specified in the *TIME* format and can be easily computed by converting the time into a millisecond representation. For Example 4 it is $2,182,980,130ms$.
With these preparations we can now define *timed action blocks*.

Definition 21 (Timed Action Block) *For $a \in Act$, $d \in \mathbb{R}$ and $q \in \{D, L, DS, SD, SL\}$ the tuple $((q, d), a)$ is a* timed action block. *We use $B_{Act,Timed}$ to denote the set of all timed action blocks and $B_{Act+Timed} := B_{Act} \cup B_{Act,Timed}$ for the set of all action blocks.*

Now let us take a closer look on the different timed action qualifiers. Let $t_{act,s}$ be the point in time, where a step $s \in Steps$ becomes active and $t_{deact,s}$ the point in time, where s becomes inactive.
An **L**-qualified action $((L, d), a) \in Blocks(s)$ will be active for a duration d (Figure 4.2: **L**), but becomes directly inactive, if the corresponding step s becomes inactive. I.e., the execution time of a is $min(d, t_{deact,s} - t_{act,s})$. Of course the action can be reset while execution. Then the execution time depends on the point in time, where the *reset* becomes active.
For a **D**-qualified action $((D, d), a) \in Blocks(s)$ the execution of a will be delayed for a duration d (Figure 4.2: **D**). The action will only be executed if the time delay is over before the step is deactivated. That means a will only be executed if $d \leq t_{deact,s} - t_{act,s}$.
We can write an **S** as prefix of **D**, **L** and get **SD** and **SL**. Actions which are **SD**-, **SL**-qualified will be executed independently from the step activation. That means an **SL**-qualified action will be executed for a duration d in either case unless the action will be reset at time point $0 < t_{reset} < d$ and a **SD**-qualified action will be executed after a delay of d time units in either case unless it will be reset before (Figure 4.2: **SL**, **SD**).

A **DS**-qualified $((DS, d), a) \in Blocks(s)$ action depends in contrast to an **SD**-qualified action on the activation time of the step. That means the corresponding action will be only set after a delay of d time units and if the step s is still active after that period of time. In both cases the actions are active until they will be reset (Figure 4.2: **DS**).

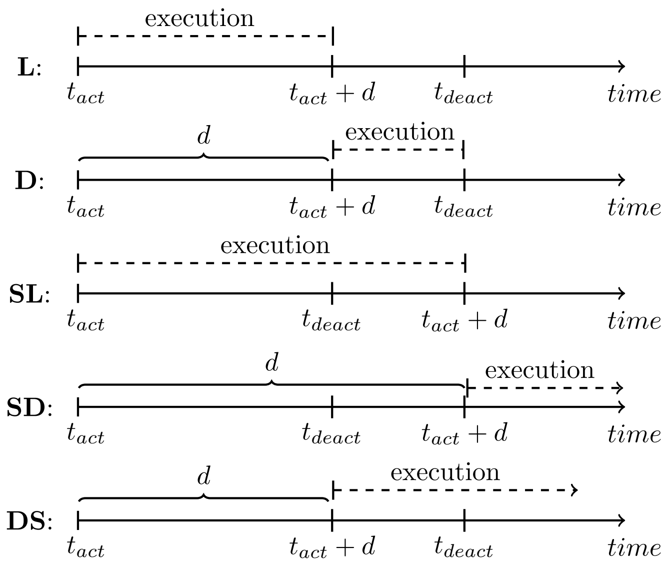


Figure 4.2: **L**, **D**, **DS**, **SD**, **SL** qualifier.

A *reset* does not influence the elapsed time of timed action qualifiers but prevents the execution of the reset action. That means, e.g., an **L**-qualified action with a duration d will not be executed after the *reset* disappears for a duration d. Instead the execution time is the difference between the duration d and the time the action was reset. In particular it is possible that an action will never be executed. This is e.g. the case if an **L**-qualified action is reset for the whole time from begin of activation until its execution duration elapses.
It is not possible to reassign another value d to a timed qualifier during execution time.

4.2 Timer/counter

To enable the time-dependent execution of actions we must introduce timers/counters, which are able to measure the elapsed time. Counters are related to steps, while timers are directly related to actions. Set actions, which are independent from the step activity will be mapped to timers, while all other actions will be mapped to counters. The reason for that is the deactivation of nested SFCs with set true flags. Set actions will be executed independently from its containing SFC and thus their timers must not be stopped if the SFC becomes inactive. In contrast, the timers of actions, which are not set, must be stopped upon the deactivation of their SFCs and possibly continued if the

containing SFC becomes active again and if its history flag is set.
So we must be able to stop the counter in case of deactivation of the maintaining SFC. If the history flag is set the counter continues after reactivation, where it was stopped. Both, timer and counter can be reset to zero. We give a formal definition for counters and timers following [BHLE04]:

Definition 22 (Timer/Counter)

- *A counter $\Theta_s^{Counter}$ for a step s is a real-valued variable. We denote the set of all counters with $\Theta_S^{Counter}$ and define a function $f : \Theta_S^{Counter} \to \{0, 1\}$ with $f(\Theta_s^{Counter}) = 1$, if the counter $\Theta_s^{Counter} \in \Theta_S^{Counter}$ is running, and $f(\Theta_s^{Counter}) = 0$, if the counter is stopped.*
- *A timer Θ_a^{Timer} for an action a is a real-valued variable. The set of all timers is denoted with Θ_A^{Timer}. We extend the function f to $f : \Theta_S^{Counter} \cup \Theta_A^{Timer} \to \{0, 1\}$ and illustrate the timer characteristic, a timer cannot be stopped, by $f(\Theta_a^{Timer}) = 1$ for all $\Theta_a^{Timer} \in \Theta_A^{Timer}$.*

Timer and Counter can be reset to zero. We write $\Theta = \Theta_S^{Counter} \cup \Theta_A^{Timer}$ for the set of all timers and counters.

4.3 SFC_{Timed} syntax

With the timed action blocks introduced in Definition 21 and timers and counters described in Definition 22 we are now able to define timed SFCs following [BHLE04].
This is an extension of the SFC$_{Set}$ syntax (Definition 11), where the function, which assigns an action block to a step, will be adapted to assign also *timed* action blocks to a step. Furthermore we extend the SFC to hold the timers and counters of time-qualified actions.

Definition 23 (SFC$_{Timed}$) *We extent the definition of SFCs (Definition 11). Let $C = (Var, \Theta, Steps, Act, s_0, Trans, Blocks, \sqsubset, \prec, Hist) \in SFC_{Timed}$ be an SFC with*

- *Var, Steps, Act, s_0, Trans, $\sqsubset$, $\prec$, Hist as in Definition 11.*
- *$Blocks : Steps \to 2^{B_{Act+Timed}}$.*
- *Θ the set of timers/counters.*

In the following, we call SFCs, which contain time-qualified actions, *timed SFCs* or SFC$_{Timed}$ to distinguish them from the foregoing defined SFCs. The set timed qualifiers **SD**, **SL** and **DS** are called *setter time* qualifier.

4.4 SFC_{Timed} semantics

We adapt the semantic of SFCs given in Section 3.3 to deal with timed action qualifiers. In the syntax definition we follow [BHLE04].
A configuration tuple for an SFC$_{Timed}$ $C = (Var, \Theta, Steps, Act, s_0, Trans, Blocks, \sqsubset, \prec, Hist)$ depends additionally on the valuation of counters and timers. Therefore we introduce the *timed configuration* $(\sigma, \upsilon, \underline{readyS}, \underline{activeS}, \underline{activeA}, \underline{storedA}, storedDA, storedLA) \in \Sigma \times 2^{\mathbb{R}} \times 2^{\overline{Steps}} \times 2^{\overline{Steps}} \times 2^{\overline{Act}} \times 2^{\overline{Act}} \times 2^{\overline{Act}} \times 2^{\overline{Act}}$ and

let $Conf^t$ be the set of all *timed configurations*.
To model time elapse, we introduce the *timed transition relation* $\rightarrow_\delta \subseteq Conf^t \times Conf^t$. Timed transitions are executed between instantaneous transitions, which we denote with $\rightarrow_a \subseteq Conf^t \times Conf^t$. We use $\rightarrow_\delta$ to adapt the valuation of the timer and extend $\rightarrow_a$ to execute actions.
A run on a SFC$_{Timed}$ consists of alternating discrete and timed transition relations. The difference between the two transition relations is that the discrete transition relation is used to execute actions and the timed transition relation changes the timer/counter values in dependence to the elapsed time. We can introduce a transition relation $\rightarrow \subseteq \rightarrow_\delta \cup \rightarrow_a$, which comprises the two transition relations.
We need two further sets (*storedLA*,*storedDA*) to store set delay (SD) and set limited actions (SL). In this way we can check for each of these actions the elapsed time at the beginning of a cycle and execute them respectively, also if their corresponding steps become inactive in the meantime. To model the time elapse we add υ to the configuration tuple, which maps rational numbers to timers and counters.

Definition 24 (Timed configuration) *Let* $c = (\sigma, \upsilon, readyS\ , activeS\ , activeA\ , storedA\ , storedDA, storedLA)$ *be a configuration of an* SFC_{Timed} C*, with*

- $\sigma, readyS\ , activeS\ , activeA\ , storedA$ *as in Definition 3,*
- υ *represents the valuation of timers and counters,*
- $storedDA \subseteq \overline{Act}$ *is the set of stored delay actions,*
- $storedLA \subseteq \overline{Act}$ *is the set of stored limited actions.*

We denote the set of all timed configurations with $Conf^t$.

In contrast to the *transition relation* of foregoing chapters, we must extend the algorithm, which collects the active actions, to deal with timed action qualifiers. Additionally we give instructions to control the timers and counters.
A counter will be reset, if the corresponding step becomes active and a timer will be reset if the corresponding action occurs in the action block of a step, which becomes active.
We will now extend the *transition relation* and introduce the *timed transition relation* in the following:

Definition 25 (Transition relation) *The transition relation* $\rightarrow_a \subseteq Conf^t \times Conf^t$ *is defined as a configuration change. For a transition* $(c, c') \in \rightarrow_a$ *with* $c := (\sigma, \upsilon, readyS\ , activeS\ , activeA\ , storedA\ , storedDA, storedLA)$ *and* $(\sigma', \upsilon', readyS', activeS', activeA', storedA', storedDA', storedLA') =: c'$*, we define*

- $readyS', \sigma'$ *as in Definition 12,*
- $(activeS', unsortedActiveA, storedA', storedDA', storedLA') = computeActiveSets_{Timed}(readyS', \emptyset, storedA, C, c, activeA\ \cap \overline{C}, storedA\ , \emptyset, \emptyset, \emptyset)$,
- $activeA' = sort(unsortedActiveA', \sqsubset)$,

- $\upsilon'(\Theta_s^{Counter}) := 0 \quad \forall s \in activeS' \backslash activeS$ *and*
  $\upsilon'(\Theta_a^{Timer}) := 0 \quad \forall a \in ((storedLA' \backslash storedLA) \cup (storedDA' \backslash storedDA))$.

The function computeActiveSets$_{Timed}$ is defined in Algorithm 4 and the function sort(...) arranges the actions descending with respect to the given order $\sqsubset$.

Let us now take a closer look at the algorithm, which collects the active actions:
For the computation of the sets *storedDA'* and *storedLA'*, we extend Algorithm 2 to deal with timed action qualifiers. We extract from the algorithm the part, which collects the active actions and put it into an external function *collectActiveActions* (Algorithm 5) to keep clarity. The remaining part results in Algorithm 4. Algorithm 5 can be described as follows:
For every timed action qualifier we add one further case to the if-branch (lines 1-15). If we get an **L**-qualified action (line 6), we check whether the corresponding step is active and how long it has been active. If the activation time of the step is shorter than the limit time d given by the action qualifier, the action will be executed. Analogously for the case **D** (line 8). The action will be executed, when the step has been active for the delay time d. If the corresponding step of a **D**-/ **L**- qualified action is deactivated in the meantime the actions will no longer be considered.
A special case is the **DS** qualifier (line 10). As **L** and **D**, it will only be considered as long as the corresponding step is active, but after a delay d the action will be added to the set of *stored* actions. That means in contrast to the **L** and **D** qualifier, the action might be executed several times independently from the step activity. We use the set of stored actions *storedA'* introduced in Section 4.4 to store an executed **DS**-qualified action and enable its periodical execution.
Independently from the step activation time are the qualifiers **SD** and **SL**, which are considered in the branches line 12 and line 14. An **SD**-qualified action will be added to the set of stored delayed actions, if it is not already part of the set, if it is not reset and if the corresponding timer valuation is lower or equal to the delay time. In contrast to the **SD** action qualifier, an **SL**-qualified action will be added to the stored limited action set, if the corresponding timer valuation is lower than the given limited time.
An **SD**-/ **SL**-qualified action will be periodically executed until it will be reset. Because we can reset an **SD**- or **SL**-qualified action before its delay/limit time is reached, we adapt the reset case (line 16) by removing actions from the *storedDA*/*storedLA* set, if they appear reset-qualified in one of the active steps. Finally, the algorithm searches for actions, which must be executed in the current cycle (line 8 - part 2). Thereby an action will be added to the set of active actions *activeA'*, if the corresponding time condition is satisfied.
Time-qualified actions will not be executed exactly in time. The reason for that is the *cyclic scanning mode*. Active actions were determined at the end of the cycle before. If, e.g., a **D**-qualified action has its execution time in the meantime of two collections of active actions, the action will be further delayed or, if it is **L**-qualified then longer executed until this second collection. In the worst case the delay time of an action elapses directly after a collection of active actions. Than the execution will delay nearly a cycle time long. Additionally, the cycle time differs in each cycle, so that we do not know whether a time-qualified action becomes active in the next cycle.
In contrast to *set*-qualified actions we add not only the action to the set of

stored actions, but also the whole action block. In this way we can add multiple timed actions with different duration times. We can check whether one of these action blocks is satisfied and if so, we can add them to the set of active actions, respectively.

For the functionality of the algorithm it is important, that the valuation of the timer will be updated in each cycle. Therefore we introduce *timed transition relations* in the following.
Counters will only be updated if their corresponding step is active, while clocks, which cannot be deactivated, will be always updated.

Definition 26 (Timed transition relation) *The timed transition relation* $\rightarrow_\delta \subseteq Conf^t \times Conf^t$ *is defined as a configuration change. For a transition* $(c, c') \in \rightarrow_\delta$ *with*
$c := (\sigma, \upsilon, readyS\ , activeS\ , activeA\ , storedA\ , storedDA, storedLA)$ *and*
$(\sigma, \upsilon', readyS\ , activeS\ , activeA\ , storedA\ , storedDA, storedLA) =: c'$ *we define*

- $\upsilon'(\Theta_s^{Counter}) := \upsilon(\Theta_s^{Counter}) + \delta,\ \forall\ s \in activeS$, *where* δ *is the execution time for the foregoing PLC cycle.*
- $\upsilon'(\Theta_a^{Timer}) := \upsilon(\Theta_a^{Timer}) + \delta,\ \forall\ \Theta_a^{Timer}$

Based on the *transition relation* and the *timed transition relation* we are now able to introduce *timed runs*:

Definition 27 (Timed run) *A* timed run *is an alternating sequence of discrete and timed transitions*
$\langle c_0 \rightarrow_{\delta_0} c_1 \rightarrow_a c'_1 ... \rightarrow_{\delta_i} c_{i+1} \rightarrow_a c'_{i+1} ...\rangle$, *where* $\delta_l \leq \delta_i \leq \delta_u$ *for all* $i \in \mathbb{N}_0$.

Now we have defined the semantics of SFC$_{Timed}$. We know the characteristics of the different timed qualifiers and how we can deal with them in SFCs.
In the following we give some more definitions for timed SFCs, which allow intuitive definitions for the transformation of SFC$_{Timed}$.

4.4.1 Additional semantic definitions for timed SFCs

As in the chapters before we provide functions, which return all actions, which are qualified with a dedicated qualifier. We do this for a local step and with consideration of nested SFCs.

Definition 28 (Specific action set function)
Let $C = (Var, \Theta, Steps, Act, s_0, Trans, Blocks, \sqsubset, \prec, Hist) \in SFC_{Timed}$ *be an SFC with configuration* $c = (\sigma, \upsilon, readyS\ , activeS\ , activeA\ , storedA\ , storedDA, storedLA) \in Conf^t$. *We define*

- $L : Steps \rightarrow 2^{Act}$ *with* $L(s) = \{a \mid (L, a) \in Blocks(s)\}$,
- $D : Steps \rightarrow 2^{Act}$ *with* $D(s) = \{a \mid (D, a) \in Blocks(s)\}$,
- $SL : Steps \rightarrow 2^{Act}$ *with* $SL(s) = \{a \mid (SL, a) \in Blocks(s)\}$,
- $SD : Steps \rightarrow 2^{Act}$ *with* $SD(s) = \{a \mid (SD, a) \in Blocks(s)\}$,
- $DS : Steps \rightarrow 2^{Act}$ *with* $DS(s) = \{a \mid (DS, a) \in Blocks(s)\}$.

Algorithm 4: $computeActiveSets_{Timed}($ $readyS', activeS, activeA, C, c, activeSFCs, storedA,$ $storedDA, storedLA, resetA)$

```
input  : $readyS', activeS, activeA,$
         $C = (Var, \Theta, Steps, Act, s_0, Trans, Blocks, \sqsubset, \prec, Hist), c,$
         $activeSFCs, storedA, storedDA, storedLA, resetA$
output: $activeS', activeA', storedA', storedDA', storedLA'$
/* Add the local active steps of $C$ to $activeS'$.              */
1  if $Hist = 1 \vee C \in activeSFCs$ then
2      $activeS' := activeS \cup (Steps \cap readyS')$;
3  else
4      $activeS' := activeS \cup \{s_0\}$;
/* Collect the local active actions and their qualifiers.   */
5  $activeA' := activeA$;
/* Stores the former set actions.                            */
6  $storedA' := storedA$;
/* Stores the former timed-set actions.                      */
7  $storedDA' := storedDA$;
8  $storedLA' := storedLA$;
/* Collect the active actions.                               */
9  $collectActiveActions(activeS', activeA', C, c, storedA, storedDA,$
           $storedLA, resetA)$;
/* Compute $activeA'$ and $activeS'$ for each active nested SFC   */
/* Edited:                                                   */
10 foreach $s \in Steps \cap activeS', b = (q, a) \in Block(s)$ do
11     if $a \in \overline{C}$ then
12         $(activeS', activeA', storedA', storedDA', storedLA') :=$
           $\mathrm{computeActiveSets_{Timed}}(readyS', activeS', activeA', a, c, activeSFCs,$
                 $storedA', storedDA', storedLA' resetA)$;
13 end
/* Add active stored actions to the active actions.          */
14 foreach $a \in storedA'$ do
15     if $a \notin activeA'$ then
16         $activeA' := activeA' \circ a$
17 end
/* Add active delayed/ stored actions to the active actions.
   */
18 foreach $b = ((q, d), a) \in storedDA'$ do
19     if $\Theta_a^{Timer} \geq d \wedge a \notin activeA'$ then
20         $activeA' := activeA' \circ a$
21 end
22 foreach $b = ((q, d), a) \in storedLA'$ do
23     if $d \geq \Theta_a^{Timer} \wedge a \notin activeA'$ then
24         $activeA' := activeA' \circ a$
25 end
26 return $(activeS', activeA', storedA', storedDA', storedLA')$;
```

Algorithm 5: *collectActiveActions*($activeS', activeA', C, c, storedA, storedDA, storedLA, resetA$)

```
   input : $activeS', activeA'$,
           $C = (Var, \Theta, Steps, Act, s_0, Trans, Blocks, \sqsubset, \prec, Hist), c$,
           $storedA, storedDA, storedLA, resetA$
1  foreach $s \in Steps, b \in Blocks(s)$ do
2      if $b = ((q, d), a)$ then
           /* Add L actions, which are not resetted.                    */
3          if
           $(q = L \wedge \Theta_s^{Counter} \leq d \wedge s \in activeS' \wedge a \notin activeA' \wedge a \notin resetA)$
           then
4              $activeA' := activeA' \circ a$;
           /* Add D actions, which are not resetted.                    */
5          else if
           $(q = D \wedge \Theta_s^{Counter} \geq d \wedge s \in activeS' \wedge a \notin activeA' \wedge a \notin resetA)$
           then
6              $activeA' := activeA' \circ a$;
           /* Add DS actions, which are not resetted.                   */
7          else if $(q = DS \wedge \Theta_s^{Counter} \geq d \wedge s \in activeS' \wedge a \notin resetA)$ then
8              $storedA' := storedA' \cup \{a\}$;
           /* Add SD actions, which are not resetted.                   */
9          else if $(q = SD \wedge \Theta_a^{Timer} \geq d \wedge s \in activeS' \wedge a \notin resetA)$ then
10             $storedDA' := storedDA' \cup \{b\}$;
           /* Add SL actions, which are not resetted.                   */
11         else if $(q = SL \wedge \Theta_a^{Timer} \leq d \wedge s \in activeS' \wedge a \notin resetA)$ then
12             $storedLA' := storedLA' \cup \{b\}$;
13     else if $b = (q, a)$ then
14         if $([(q = exit \wedge s \in source(taken(C, c))) \vee (q = entry \wedge s \in$
           $target(taken(C, c))) \vee (q = do \wedge s \in activeS')] \wedge a \notin activeA' \wedge \notin$
           $resetA)$ then
15             $activeA' := activeA' \circ a$;
           /* Add set actions, which are not resetted.                  */
16         else if $(q = set \wedge s \in activeS' \wedge a \notin resetA)$ then
17             $storedA' := storedA' \cup \{a\}$;
18         else if $(q = reset \wedge s \in activeS')$ then
19             $resetA := resetA \cup \{a\}$;
               if $a \in activeA'$ then
20                 $activeA' := activeA' \backslash a$
21             if $a \in storedA'$ then
22                 $storedA' := storedA' \backslash \{a\}$
23             if $\exists\, t : ((SD, t), a) \in storedDA'$ then
24                 $storedDA' := storedDA' \backslash \{b = ((SD, t'), a) \mid \forall\, t' \in \mathbb{R}\}$
25             if $\exists\, t : ((SL, t), a) \in storedLA'$ then
26                 $storedLA' := storedLA' \backslash \{b = ((SL, t'), a) \mid \forall\, t' \in \mathbb{R}\}$
27 end
```

We introduce the following functions, which consider also nested SFCs.

- $L_{Nested} : Conf^t \to 2^{Act}$ *with* $L_{Nested}(c) = \bigcup_{s \in activeS} L(s)$,
- $D_{Nested} : Conf^t \to 2^{Act}$ *with* $D_{Nested}(c) = \bigcup_{s \in activeS} D(s)$,
- $SL_{Nested} : Conf^t \to 2^{Act}$ *with* $SL_{Nested}(c) = \bigcup_{s \in activeS} SL(s)$,
- $SD_{Nested} : Conf^t \to 2^{Act}$ *with* $SD_{Nested}(c) = \bigcup_{s \in activeS} SD(s)$,
- $DS_{Nested} : Conf^t \to 2^{Act}$ *with* $DS_{Nested}(c) = \bigcup_{s \in activeS} DS(s)$.

4.5 $HSFC_{Timed}$

We have formalized the syntax and semantics of time-qualified SFCs. We get a HSFC by assigning conditional ODEs to steps, as we have seen in Section 2.3. The syntax and semantics of timed SFC can be easily extended to timed hybrid sequential function charts. We can transform a given timed SFC into a timed HSFC according to Definitions 2.3.1 and 2.3.2.

4.6 $(H)SFC_{Timed} \to$ Hybrid automaton

We will now face the challenge to transform a timed SFC into a hybrid automaton. This will allow us the verification of timed SFCs.
Because of the different behaviors and characteristics of the timed action qualifiers, we give for each qualifier separate transformation instructions. The intent is to ensure clarity and to reduce the amount of operations. So we must only apply these transformations, where the corresponding action qualifier occurs in the SFC. In other words, if an SFC contains a dedicated timed action qualifier, the corresponding transformation can be applied on it independently from not occurring timed action qualifiers. Thereby the order of transformations is arbitrary (Figure 4.3).

$$\begin{tikzcd}
 & \text{HA + SD} \arrow[rr, <->] \arrow[dl, <->] \arrow[drrr, <->] \arrow[ddr, <->] & & \text{HA + SL} \arrow[dlll, <->] \arrow[dr, <->] \arrow[ddl, <->] & \\
\text{HA + L} \arrow[rrrr, <->] \arrow[drr, <->] & & & & \text{HA + D} \arrow[dll, <->] \\
 & & \text{HA + DS} & &
\end{tikzcd}$$

Figure 4.3: SFC$_{Timed}$ transformation.

The whole process of transforming a timed SFC can be described as follows: If we have a timed SFC, we begin by transforming it into a hybrid automaton, as described in Chapter 2 or if it contains stored actions, as described in Chapter 3 (Figure 4.1).
On the resulting automaton we apply those timed transformations in arbitrary order, where the corresponding timed action qualifier is part of the SFC. Finally we get an automaton, which is reachability-equivalent to the corresponding timed

SFC.
Before we start with the transformation instructions and their formal definitions, we introduce a timer and some more functions.

4.6.1 Timer automaton

To execute actions limited or with a specific delay, we must measure the elapsed time. Therefore we introduce a *timer* variable t_{timer}, which will be updated as long as the program will be executed. This variable is globally provided to each step and action by a timer automaton (Figure 4.4), so that we can use it to determine active actions.
To minimize the number of timers in the resulting hybrid automaton, we do not create an associated timer for each timed action qualifier. Instead we store the timestamp of a timed action, which becomes active and add the current time t_{timer} to the time duration d given by the timed qualifier.
The advantage is that we only need to increase one timer variable per time unit and because we do this in the activities of locations, we only need to add one activity to each location. In this way we save a lot of operations, especially in practical use, but introduce variables for storing the timestamps.
We restrict our definitions to timed SFCs, which do not contain parallel branches and nested SFCs. As a consequence we must not consider history flags and therefore we need not to distinguish between timers and counters.

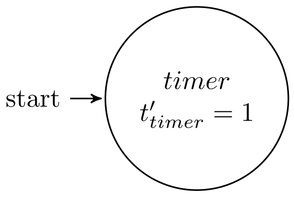


Figure 4.4: Timer automaton.

To know whether an action must be executed we follow our concept of setter variables and adapt it to deal with time values. The *setter* variables will not only contain the values one and zero, but real numbers, which can be used to determine whether a delay or limit is elapsed.
Following this approach we create *setter* variables and transitions for *setter time* qualifiers as we have seen in Section 3 with adapted *setter* actions. We will further add the timer variable t_{timer} of the timer automaton (Figure 4.4) and for **DS**-qualified actions we add *Setter* variables ξ_a to enable the execution until a reset appears.
So let us define the set of variables Var_{Timed}, which are added to the hybrid automaton to deal with timed SFCs.

Definition 29 (Var_{Timed})
$$\begin{aligned} Var_{Timed} := & \{t_{timer}\} \\ & \cup \{t_b \mid b = (q,a) \in Blocks(s), s \in Steps \wedge q \in \{L, D, SD, SL, DS\}\} \\ & \cup \{\xi_a \mid b = (q,a) \in Blocks(s), s \in Steps \wedge q \in \{DS\}\} \end{aligned}$$

We use control actions to change the valuation of the variables. Thereby we distinguish two actions. a_t^{stamp} copies the current time t_{timer} into the timer

variable t. We do this at step activation and know in this way its activation time. This can be used in the following to execute e.g. a delayed action.
The *reset* control action a_t^{reset} assigns the value -1 to a timer variable t. Since negative delays or execution durations are senseless, we use -1 as special key number to check whether the action was reset.
We define the actions, which store and reset the timestamps:

Definition 30 (Timer control actions) *For each timed variable* $t \in Var_{Timed}$ *we define actions* $Act := \{a_t^{stamp}, a_t^{reset}\} \cup Act$ *with:*

- $a_t^{stamp} : t := t_{timer}$
- $a_t^{reset} : t := -1$

Now, we will introduce some additional functions, which we use to change the guards and action assignments of already existing transitions. As described before we start with the hybrid automaton, which is the transformation of an SFC or SFC$_{Set}$, if we want to transform a SFC$_{Timed}$. We then change the automaton so that it will handle also timed qualifiers. Therefore we need some functions, which allow the modification of the automaton in an understandable way.

4.6.2 HA modification functions

Since we adapt the transition guards and assignments of transitions, we need a helper function *combine(...)*, which allows us to extend a transition from the set *trans* with a guard g, an action a and with the negated guard $\neg g$. We assume a call-by-reference characteristic, that means the automaton will be directly manipulated in the function.
In other words: We take all transitions specified in the set *trans* and change them in the automaton H, by extending the transition guard by g and adding one action a. Furthermore we add one transition with the negated guard, but without any additional assignments.

Definition 31 ($\boldsymbol{combine}(H, \boldsymbol{trans}, g, \boldsymbol{act})$ function) *We define a function* $combine : Aut \times 2^{Edge} \times G^{Var} \times Act_{SFC} \rightarrow Aut$ *with* $combine(H, trans, g, a) = H'$ *for* $H = (Loc, Edge, Act, Inv, L, sync, Init, \sqsubset)$ *and* $H' = (Loc, Edge', Act, Inv, L, sync, Init, \sqsubset)$. *We define* $Edge'$ *as follows:*

– $Edge' = (Edge \backslash trans)$
$\cup\{(s, (\overline{\mu}_1, \overline{\mu}'_1), s') \mid (s, (\mu_1, \mu'_1), s') \in (trans \cap Edge)\}$
$\cup\{(s, (\overline{\mu}_2, \mu'_2), s') \mid (s, (\mu_2, \mu'_2), s') \in (trans \cap Edge)\}$,
where $\exists v \in V, \exists a_1 ... \exists a_n \in Var$: $\{v(a_1), ..., v(a_n)\} = \mu'_1$ *with* $v(a_1) \sqsubset ... \sqsubset v(a_n)$ *and*
$\forall\, a'_l \in (act^{-1}(\mathbb{R}) \cap Var), \exists\, a^i_l, a^j_l \in Var : \{v(a_1), ..., v(a^i_l), v(a'_l), v(a^j_l), ..., v(a_n)\} = \overline{\mu}'_1$ *with* $v(a_1) \sqsubset ... \sqsubset v(a^i_l) \sqsubset v(a'_l) \sqsubset v(a^j_l) \sqsubset ... \sqsubset v(a_n)$ *and*
$\forall g' \in G^{Var} : \mu_1 \models g' \Rightarrow \overline{\mu}_1 \models g' \wedge g$ *and*
$\forall g' \in G^{Var} : \mu_2 \models g' \Rightarrow \overline{\mu}_2 \models g' \wedge \neg g$.

We need timed control actions to manipulate timer variables. Their mapping to transition assignments depends only on the action qualifiers of the target location, that means those qualifiers which occur in the corresponding step of an SFC. In particular timed control actions are independent from any transition guard so we need a function, which adds only an action to the transition assignments without changing the transition guard. The following function will add an action a to transitions of a set *trans*. As in the definition of the function before we will assume a call-by-reference behavior. That means we manipulate the hybrid automaton directly.

Definition 32 ($\boldsymbol{addAction}(H, \boldsymbol{trans}, \boldsymbol{act})$ function) *We define a function $addAction : Aut \times 2^{Edge} \times Act_{SFC} \rightarrow Aut$ with $addAction(H, trans, act) = H'$, where $H = (Loc, Edge, Act, Inv, L, sync, Init, \sqsubset)$ and $H' = (Loc, Edge', Act, Inv, L, sync, Init, \sqsubset)$. We define $Edge'$ as follows:*

– $Edge' = (Edge \backslash trans)$
$\cup \{(s, (\mu_1, \overline{\mu}'_1), s') \mid (s, (\mu_1, \mu'_1), s') \in (trans \cap Edge)\}$
where $\exists v \in V, \exists a_1 ... \exists a_n \in Var$*:* $\{v(a_1), ..., v(a_n)\} = \mu'_1$ *with* $v(a_1) \sqsubset ... \sqsubset v(a_n)$ *and*
$\forall\, a'_l \in (act^{-1}(\mathbb{R}) \cap Var), \exists\, a^i_l, a^j_l \in Var : \{v(a_1), ..., v(a^i_l), v(a'_l), v(a^j_l), ..., v(a_n)\} = \overline{\mu}'_1$ *with* $v(a_1) \sqsubset ... \sqsubset v(a^i_l) \sqsubset v(a'_l) \sqsubset v(a^j_l) \sqsubset ... \sqsubset v(a_n)$.

4.6.3 The transformation

Based on the preparations we did in the chapters before, we will now give instructions to model timed action qualifiers in hybrid automata.
We begin with the transformation instructions for the **L** and the **D** qualifiers. We will then define the transformation of **DS**-qualified actions, which are particular because we need *setter* variables to determine if the action must be executed. At the end we will introduce the *setter time* action qualifiers **SL** and **SD**.
We begin by defining the transitions $Edge_L$, which will be added to the hybrid automaton to model **L**-qualified actions.

Transformation of L-qualified actions

For a location, where the corresponding step contains an **L**-qualified action, we replace each self loop by two transitions. The first will be taken if the activation time is in the limit and the corresponding timestamp variable is not equal -1. That will execute the corresponding action, while the other one will be taken if the limit is overdue or the corresponding timestamp variable is -1. In this case the corresponding action may no longer be executed. Incoming transitions of the step will be extended to timestamp actions, so that we can add the qualifier given time d to the timestamp and compare it with the global time variable t_{timer}, if an **L**-qualified action becomes active.

Definition 33 ($Edge_L(H, C)$) *We define a function $Edge_L : Aut \times SFC_{Set+Timed} \rightarrow Aut$ with $Edge_L(H, C) = H'$, where $C = (Var, \Theta, Steps, Act_{SFC}, s_0, Trans, Blocks, \sqsubset_{SFC}, \prec, Hist)$ is an SFC and $H = (Loc, Var, Edge, Act, Inv, Init, \sqsubset)$ its transformation. We construct H' as follows:*

– *Let* $\{b_1 = ((L, d_1), a_1), ..., b_n = ((L, d_n), a_n)\} = \{b = ((L, d), a) \mid b \in B_{Act}\}$ *be the **L**-qualified actions of* C*:*

  - $H''_1 = addAction(combine(H, \{t = (s, \mu, s) \mid t \in Edge \wedge b_1 \in Blocks(s) \wedge sync(t) = action\_sync\}, t_{b_1} + d_1 \geq t_{timer} \wedge t_{b_1} \neq -1, a_1), \{t = (s', \mu, s) \mid t \in Edge \wedge b_1 \in Blocks(s) \wedge s \neq s'\}, a^{stamp}_{t_{b_1}})$
  - $H''_i = addAction(combine(H''_{i-1}, \{t = (s, \mu, s) \mid t \in Edge \wedge b_i \in Blocks(s) \wedge sync(t) = action\_sync\}, t_{b_i} + d_i \geq t_{timer} \wedge t_{b_i} \neq -1, a_i), \{t = (s', \mu, s) \mid t \in Edge \wedge b_i \in Blocks(s) \wedge s' \neq s\}, a^{stamp}_{t_{b_i}})$ *for* $i \in \{2, ..., n\}$

- *Let* $\{(b_1 = ((R, d_1), a_1), ..., b_m = ((R, d_m), a_m)\} = \{b = ((R, d), a) \mid b \in B_{Act}\}$ *be the* ***R****-qualified actions of* C*:*
  - $H'''_1 = addAction(H''_n, \{t = (s', \mu, s) \mid t \in Edge \wedge b_1 \in Blocks(s) \wedge s \neq s'\}, a^{reset}_{t_{b_1}})$
  - $H'''_j = addAction(H'''_{j-1}, \{t = (s', \mu, s) \mid t \in Edge \wedge b_j \in Blocks(s) \wedge s \neq s'\}, a^{reset}_{t_{b_j}})$*, for* $j \in \{2, ..., m\}$

- $H' := H'''_m$

We give an example in Figure 4.5, where we add one transition, which will execute the timed action and one transition, which can be taken if the limit time is elapsed. Timer control actions are added to the incoming transitions to update the timer variables.

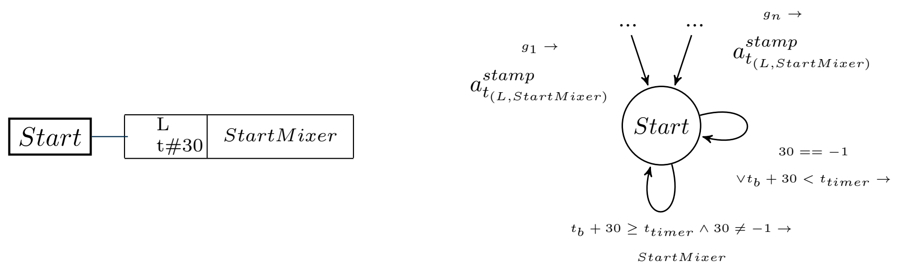

Start
L
t#30
StartMixer
...
...
$g_1 \rightarrow$
$a^{stamp}_{t_{(L,StartMixer)}}$
$g_n \rightarrow$
$a^{stamp}_{t_{(L,StartMixer)}}$
Start
$30 == -1$
$\vee t_b + 30 < t_{timer} \rightarrow$
$t_b + 30 \geq t_{timer} \wedge 30 \neq -1 \rightarrow$
$StartMixer$

Figure 4.5: L qualifier transformation.

We will now consider the transitions, which we add to the hybrid automaton to deal with delayed actions.

Transformation of D-qualified actions

As in Definition 33 we replace each existing self loop of steps, which contains **D**-qualified actions, with two transitions, where the first one executes the corresponding action, if the delay is reached. A second one can be taken in the meantime. This transition will not execute the time-qualified action. Incoming transitions will be extended to an action, which sets the timestamp. As before the reset is done by setting the timestamp to -1.

Definition 34 ($Edge_D(H, C)$**)** *We define a function* $Edge_D : Aut \times SFC_{Set+Timed} \rightarrow Aut$ *with* $Edge_D(H, C) = H'$*, where* $C = (Var, \Theta, Steps, Act_{SFC}, s_0, Trans, Blocks, \sqsubset_{SFC}, \prec, Hist)$ *is an SFC and* $H = (Loc, Var, Edge, Act, Inv, Init, \sqsubset)$ *its transformation. We construct* H' *as follows:*

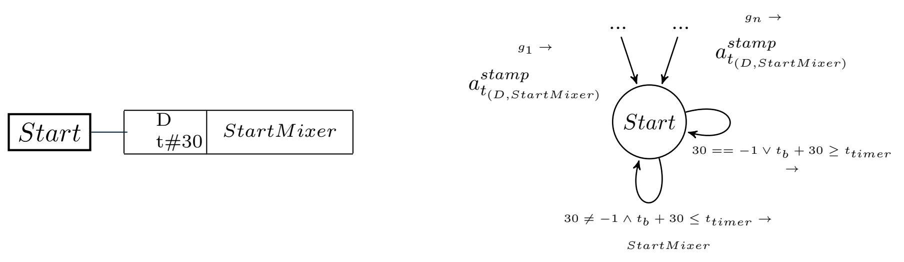
- *Let* $\{b_1 = ((D, d_1), a_1), ..., b_n = ((D, d_n), a_n)\} = \{b = ((D, d), a) \mid b \in B_{Act}\}$ *be the* ***D****-qualified actions of* C*:*

  - $H_1'' = addAction(combine(H, \{t = (s, \mu, s) \mid t \in Edge \wedge b_1 \in Blocks(s) \wedge sync(t) = action\_sync\}, t_{b_1} + d_1 \leq t_{timer} \wedge t_{b_1} \neq -1, a_1), \{t = (s', \mu, s) \mid t \in Edge \wedge b_1 \in Blocks(s) \wedge s \neq s'\}, a_{t_{b_1}}^{stamp})$
  - $H_i'' = addAction(combine(H_{i-1}'', \{t = (s, \mu, s) \mid t \in Edge \wedge b_i \in Blocks(s) \wedge sync(t) = action\_sync\}, t_{b_i} + d_i \leq t_{timer} \wedge t_{b_i} \neq -1, a_i), \{t = (s', \mu, s) \mid t \in Edge \wedge b_i \in Blocks(s) \wedge s' \neq s\}, a_{t_{b_i}}^{stamp})$ *for* $i \in \{2, ..., n\}$

- *Let* $\{(b_1 = ((R, d_1), a_1), ..., b_m = ((R, d_m), a_m)\} = \{b = ((R, d), a) \mid b \in B_{Act}\}$ *be the* ***R****-qualified actions of* C*:*
  - $H_1''' = addAction(H_n'', \{t = (s', \mu, s) \mid t \in Edge \wedge b_1 \in Blocks(s) \wedge s \neq s'\}, a_{t_{b_1}}^{reset})$
  - $H_j''' = addAction(H_{j-1}''', \{t = (s', \mu, s) \mid t \in Edge \wedge b_j \in Blocks(s) \wedge s \neq s'\}, a_{t_{b_j}}^{reset})$*, for* $j \in \{2, ..., m\}$

- $H' := H_m'''$

As in the example before (Example 4.5) we introduce two transition, where the first will be taken if the time condition is satisfied and the second otherwise. Also in this case we add time control actions to change the valuation of the timer variables.



Start
D
t#30
$StartMixer$
...
...
$g_1 \rightarrow$
$a_{t(D,StartMixer)}^{stamp}$
$g_n \rightarrow$
$a_{t(D,StartMixer)}^{stamp}$
$Start$
$30 == -1 \vee t_b + 30 \geq t_{timer}$
$\rightarrow$
$30 \neq -1 \wedge t_b + 30 \leq t_{timer} \rightarrow$
$StartMixer$

Figure 4.6: D qualifier transformation.

Now we will consider the transformation of **DS**-qualified actions.

Transformation of DS-qualified actions

To deal with *setter timed* action qualifier, we must adapt the hybrid automaton more globally. For each **DS** qualifier we apply our *setter* qualifier approach that we have introduced in Section 3. The only difference is that we use timer depended transitions to enable a *setter* variable.
We will take a look on transitions, which must be added, if a step contains **DS**-qualified actions. If we remember the characteristics of the **DS** qualifier, we know that the corresponding action will be executed until a reset appears. That means if the step, which contains a **DS**-qualified action, is active as long as the time delay, the corresponding action will be executed and the related *setter* variable will be mapped to true.
Therefore we replace each self loop of a location, where the corresponding step in the SFC$_{Timed}$ contains a **DS**-qualified action, with two other transitions. These are nearly equal to the transitions regarded in Definition 34, with the only difference that we also set the corresponding *setter* variable. A reset will set the

timestamp variable to -1 and the corresponding setter variable to false.
In this way we can reset the **DS**-qualified action, while the delay is not elapsed.

Definition 35 ($Edge_{DS}(H, C)$) *We define a function* $Edge_{DS} : Aut \times SFC_{Set+Timed} \rightarrow Aut$ *with* $Edge_{DS}(H, C) = H'$, *where* $C = (Var, \Theta, Steps, Act_{SFC}, s_0, Trans, Blocks, \sqsubset_{SFC}, \prec, Hist)$ *is an SFC and* $H = (Loc, Var, Edge, Act, Inv, Init, \sqsubset)$ *its transformation. We construct* H' *as follows:*

- *Let* $\{b_1 = ((DS, d_1), a_1), ..., b_n = ((DS, d_n), a_n)\} = \{b = ((DS, d), a) \mid b \in B_{Act}\}$ *be the* ***DS****-qualified actions of* C*:*
  - $H_1'' = combine(H, \{t = (s, (v, v'), s') \mid t \in Edge \wedge \forall g \in G^{Var} : v \models g \Rightarrow v \not\models g \wedge \xi_{a_1}\}, \xi_{a_1}, a_1)$
  - $H_k'' = combine(H_{k-1}, \{t = (s, (v, v'), s') \mid t \in Edge \wedge \forall g \in G^{Var} : v \models g \Rightarrow v \not\models g \wedge \xi_{a_k}\}, \xi_{a_k}, a_k)$ *for* $k \in \{2, ..., n\}$
  - $H_1''' = addAction(combine(H_n'', \{t = (s, \mu, s) \mid t \in Edge \wedge b_1 \in Blocks(s) \wedge sync(t) = action\_sync\}, t_{b_1} + d_1 \leq t_{timer} \wedge t_{b_1} \neq -1, \{a_1, a^{set}_{\xi_{a_1}}\}), \{t = (s', \mu, s) \mid t \in Edge \wedge b_1 \in Blocks(s) \wedge s \neq s'\}, a^{stamp}_{t_{b_1}})$
  - $H_i''' = addAction(combine(H_{i-1}''', \{t = (s, \mu, s) \mid t \in Edge \wedge b_i \in Blocks(s) \wedge sync(t) = action\_sync\}, t_{b_i} + d_i \leq t_{timer} \wedge t_{b_i} \neq -1, \{a_i, a^{set}_{\xi_{a_i}}\}), \{t = (s', \mu, s) \mid t \in Edge \wedge b_i \in Blocks(s) \wedge s' \neq s\}, a^{stamp}_{t_{b_i}})$ *for* $i \in \{2, ..., n\}$
- *Let* $\{(b_1 = ((R, d_1), a_1), ..., b_m = ((R, d_m), a_m)\} = \{b = ((R, d), a) \mid b \in B_{Act}\}$ *be the* ***R****-qualified actions of* C*:*
  - $H_1'''' = addAction(H_n''', \{t = (s', \mu, s) \mid t \in Edge \wedge b_1 \in Blocks(s) \wedge s \neq s'\}, \{a^{reset}_{t_{b_1}}, a^{reset}_{\xi_{a_1}}\})$
  - $H_j'''' = addAction(H_{j-1}'''', \{t = (s', \mu, s) \mid t \in Edge \wedge b_j \in Blocks(s) \wedge s \neq s'\}, \{a^{reset}_{t_{b_j}}, a^{reset}_{\xi_{a_j}}\})$, *for* $j \in \{2, ..., m\}$
- $H' := H_m'''$

For a **DS**-qualified action (Figure 4.7) we get a hybrid automaton with transitions to handle the timer and the set actions (Figure 4.8). In particular we recognize that we have in addition to the time condition also always the *setter* variable.

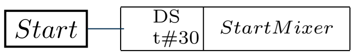


Figure 4.7: DS-qualified action.

We will face the transformation of *set*-time-qualified actions in the following.

Transformation of SL-qualified actions

The action qualifiers **SD** and **SL** have more global characteristics. The execution of those actions is independent from step activity and therefore we need to adapt all transitions in the automaton. Let us begin by defining the transitions for **SL**-qualified actions. An **SL**-qualified action must be executed until the

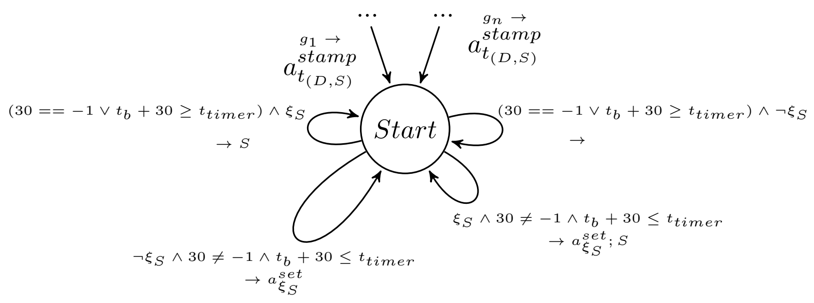

...
...
$g_1 \to a_{t_{(D,S)}}^{stamp}$
$g_n \to a_{t_{(D,S)}}^{stamp}$
$Start$
$(30 == -1 \vee t_b + 30 \geq t_{timer}) \wedge \xi_S \to S$
$(30 == -1 \vee t_b + 30 \geq t_{timer}) \wedge \neg\xi_S \to$
$\xi_S \wedge 30 \neq -1 \wedge t_b + 30 \leq t_{timer} \to a_{\xi_S}^{set}; S$
$\neg\xi_S \wedge 30 \neq -1 \wedge t_b + 30 \leq t_{timer} \to a_{\xi_S}^{set}$

Figure 4.8: DS qualifier transformation with S := StartMixer.

timer reaches the limit or the action will be reset. Thereby the execution is independent from step activation times.
We split each existing transition into two transitions, where one of them executes the **SL**-qualified action and the other one can be taken, if the limit time elapses. If a step contains an **SL**-qualified action, the incoming transitions of the step will store the timestamp. If it contains a reset we add a_t^{reset} to the incoming transitions assignments. In this way the timestamp variable is set to one and the **SL** execution condition $t + d \geq t_{timer}$ will never hold.

Definition 36 ($Edge_{SL}(H, C)$) *We define a function $Edge_{SL} : Aut \times SFC_{Set+Timed} \to Aut$ with $Edge_{SL}(H, C) = H'$, where $C = (Var, \Theta, Steps, Act_{SFC}, s_0, Trans, Blocks, \sqsubset_{SFC}, \prec, Hist)$ is an SFC and $H = (Loc, Var, Edge, Act, Inv, Init, \sqsubset)$ its transformation. We construct H' as follows:*

- *Let $\{b_1 = ((SL, d_1), a_1), ..., b_n = ((SL, d_n), a_n)\} = \{b = ((SL, d), a) \mid b \in B_{Act}\}$ be the **SL**-qualified actions of C:*
  - *$H_1'' = addAction(combine(H, \{t \mid t \in Edge \wedge sync(t) = action\_sync\}, t_{b_1} + d_1 \geq t_{timer} \wedge t_{b_1} \neq -1, a_1), \{t = (s', \mu, s) \mid t \in Edge \wedge b_1 \in Blocks(s) \wedge s \neq s'\}, a_{t_{b_1}}^{stamp})$*
  - *$H_i'' = addAction(combine(H_{i-1}'', \{t \mid t \in Edge \wedge sync(t) = action\_sync\}, t_{b_i} + d_i \geq t_{timer} \wedge t_{b_i} \neq -1, a_i), \{t = (s', \mu, s) \mid t \in Edge \wedge b_i \in Blocks(s) \wedge s' \neq s\}, a_{t_{b_i}}^{stamp})$ for $i \in \{2, ..., n\}$*
- *Let $\{(b_1 = ((R, d_1), a_1), ..., b_m = ((R, d_m), a_m)\} = \{b = ((R, d), a) \mid b \in B_{Act}\}$ be the **R**-qualified actions of C:*
  - *$H_1''' = addAction(H_n'', \{t = (s', \mu, s) \mid t \in Edge \wedge b_1 \in Blocks(s) \wedge s \neq s'\}, a_{t_{b_1}}^{reset})$*
  - *$H_j''' = addAction(H_{j-1}''', \{t = (s', \mu, s) \mid t \in Edge \wedge b_j \in Blocks(s) \wedge s \neq s'\}, a_{t_{b_j}}^{reset})$, for $j \in \{2, ..., m\}$*
- *$H' := H_m'''$*

We replace each already existing transition by two adapted ones, where the first will be taken if the time condition holds and thus the corresponding action must be executed, while the second transition will be taken if the limit time elapses. Timer control actions will be only added to those locations, where the corresponding step in the SFC contains **SL** or **R**-qualified actions.

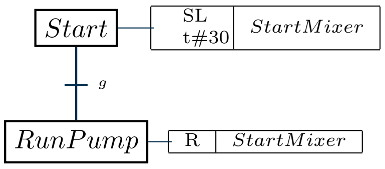


Figure 4.9: SL-qualified action.

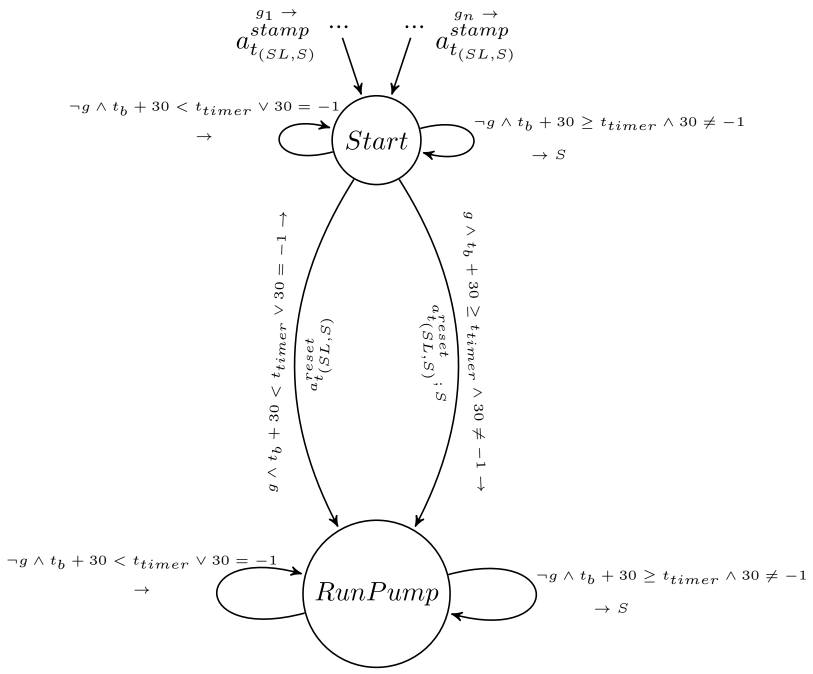

$g_1 \to a^{stamp}_{t_{(SL,S)}}$
$g_n \to a^{stamp}_{t_{(SL,S)}}$
Start
RunPump
$\neg g \wedge t_b + 30 < t_{timer} \vee 30 = -1 \to$
$\neg g \wedge t_b + 30 \geq t_{timer} \wedge 30 \neq -1 \to S$
$g \wedge t_b + 30 < t_{timer} \vee 30 = -1 \to a^{reset}_{t_{(SL,S)}}$
$g \wedge t_b + 30 \geq t_{timer} \wedge 30 \neq -1 \to a^{reset}_{t_{(SL,S)}}; S$

Figure 4.10: SL qualifier transformation with S:= StartMixer.

Transformation of SD-qualified actions

The transitions for the **SD** qualifier are very similar to the transformation of **SL**-qualified actions. In this case an action will be delayed executed until it will be reset. To check whether the timer reaches the delay we use a global timestamp variable. It is set in a location where the corresponding step in the SFC$_{Timed}$ contains an **SD**-qualified action and is reset to -1 in the incoming transitions of those locations, where the corresponding step in the SFC$_{Timed}$ contains the action **R**-qualified.
So let us define $Edge_{SD}$. We add one transition, which executes the delayed action and adapt the incoming actions of a step, which contains **SD**-qualified actions, so that the corresponding timestamp variable will be set. The reset is done by setting the corresponding timestamp variable to -1. Because we check in our execution guard also for $t \neq -1$, we can reset the action before the delay ended.

Definition 37 **(**$Edge_{SD}(H, C)$**)** *We define a function* $Edge_{SD} : Aut \times SFC_{Set+Timed} \to Aut$ *with* $Edge_{SD}(H, C) = H'$*, where* $C =$

$(Var, \Theta, Steps, Act_{SFC}, s_0, Trans, Blocks, \sqsubset_{SFC}, \prec, Hist)$ is an SFC and $H = (Loc, Var, Edge, Act, Inv, Init, \sqsubset)$ its transformation. We construct H' as follows:

- *Let $\{b_1 = ((SD, d_1), a_1), ..., b_n = ((SD, d_n), a_n)\} = \{b = ((SD, d), a) \mid b \in B_{Act}\}$ be the **SD**-qualified actions of C:*
  - *$H''_1 = addAction(combine(H, \{t \mid t \in Edge \wedge sync(t) = action\_sync\}, t_{b_1} + d_1 \leq t_{timer} \wedge t_{b_1} \neq -1, a_1), \{t = (s', \mu, s) \mid t \in Edge \wedge b_1 \in Blocks(s) \wedge s \neq s'\}, a^{stamp}_{t_{b_1}})$*
  - *$H''_i = addAction(combine(H''_{i-1}, \{t \mid t \in Edge \wedge sync(t) = action\_sync\}, t_{b_i} + d_i \leq t_{timer} \wedge t_{b_i} \neq -1, a_i), \{t = (s', \mu, s) \mid t \in Edge \wedge b_i \in Blocks(s) \wedge s' \neq s\}, a^{stamp}_{t_{b_i}})$ for $i \in \{2, ..., n\}$*
- *Let $\{(b_1 = ((R, d_1), a_1), ..., b_m = ((R, d_m), a_m)\} = \{b = ((R, d), a) \mid b \in B_{Act}\}$ be the **R**-qualified actions of C:*
  - *$H'''_1 = addAction(H''_n, \{t = (s', \mu, s) \mid t \in Edge \wedge b_1 \in Blocks(s) \wedge s \neq s'\}, a^{reset}_{t_{b_1}})$*
  - *$H'''_j = addAction(H'''_{j-1}, \{t = (s', \mu, s) \mid t \in Edge \wedge b_j \in Blocks(s) \wedge s \neq s'\}, a^{reset}_{t_{b_j}})$, for $j \in \{2, ..., m\}$*
- *$H' := H'''_m$*

As in the transformation of a **SL**-qualified action, we replace each already existing transition by two new transitions (Figure 4.12). The timed control actions will be added to transitions as seen in the examples before.

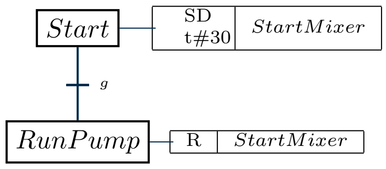
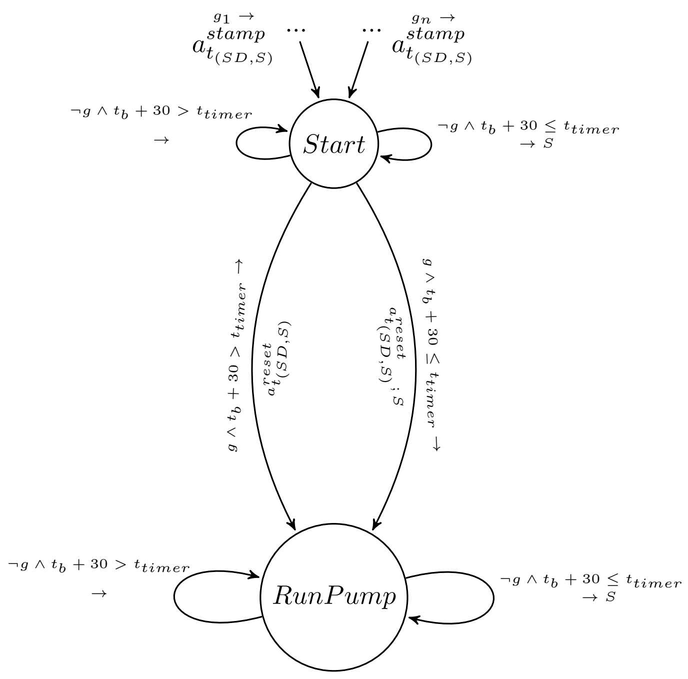


Figure 4.11: SD-qualified action.

Finally we can define the transformation of an SFC$_{Timed}$, with arbitrary timed action qualifiers. The application order of the different qualifier transformations is arbitrary. It is also possible to omit transformations, if the corresponding timed action qualifier does not occur in a given SFC$_{Timed}$.

Definition 38 (SFC → Hybrid automaton) *We extent Definition 20. Let $C = (Var, \Theta, Steps, Act, s_0, Trans, Blocks, \sqsubset, \prec, Hist) \in SFC_{Timed}$ be a timed SFC without parallel transitions and without nested SFCs and $H' = (Loc, Var_H, Edge, Act_H, Inv, Init)$ its transformation into a hybrid automaton with*

- *$Var_H = Var \cup Var_{Timed}$,*
- *Loc, Inv and Init are defined as in Definition 9,*
- *Act_H the actions extended with the control actions of Definition 30 and*
- *$Edge := Edge_{\varsigma_1}(Edge_{\varsigma_2}(Edge_{\varsigma_3}(Edge_{\varsigma_4}(Edge_{\varsigma_5}(H', C), C), C), C), C)$ with $\varsigma_i \in \{L, D, SL, SD, DS\}, i \in \{1, ..., 5\}$ and H' as defined in Definition 9.*



$g_1 \rightarrow a^{stamp}_{t(SD,S)}$
...
...
$g_n \rightarrow a^{stamp}_{t(SD,S)}$
$\neg g \wedge t_b + 30 > t_{timer} \rightarrow$
Start
$\neg g \wedge t_b + 30 \leq t_{timer} \rightarrow S$
$g \wedge t_b + 30 > t_{timer} \rightarrow a^{reset}_{t(SD,S)}$
$g \wedge t_b + 30 \leq t_{timer} \rightarrow a^{reset}_{t(SD,S)}; S$
$\neg g \wedge t_b + 30 > t_{timer} \rightarrow$
RunPump
$\neg g \wedge t_b + 30 \leq t_{timer} \rightarrow S$

Figure 4.12: SD qualifier transformation with S:= StartMixer.

4.6.4 Complexity consideration

As for the transformation of *setter* SFCs, we will now consider the space complexity of the transformations of timed SFCs. Thereby we start with a hybrid automaton and analyses the changes in consequence of the transformations. We will consider the transformation for each timed action qualifier separate and give a comparison at the end of this chapter.

Analysis of L-qualified actions

Let H be a hybrid automaton with n locations, m *action_sync* labeled transitions, k variables and let q be the number of **L**-qualified actions.
The transformation will not change the number of locations, so that the resulting automaton has also n locations. The number of variables increases by q, because we add for each **L**-qualified action one variable, which stores the timestamp. We get $k + q$ variables in the resulting automaton.
The number of additional transitions depends on the assignment of **L**-qualified actions to steps. An **L**-qualified action will duplicate the *action_sync* labeled self loops of the corresponding location. In the worst case all **L**-qualified actions will be assigned to one location. In this case we must duplicate all self loops q times. We get $O(2^q + m - 1)$ transitions, where $m - 1$ are the transitions, which were not splitted.

Analysis of D-qualified actions

Let H be a hybrid automaton with n locations, m *action_sync* labeled transitions, k variables and let q be the number of **D**-qualified actions.
As in the transformation of **L**-qualified actions the number of locations will not change and the number of variables will be increased by $k + q$, since we need for each **D**-qualified action one variable, which stores the timestamp. Once again we must duplicate the *action_sync* labeled self loops of a location for each **D**-qualified action, which is contained in the corresponding step in the SFC. With the same argument as for **L**-qualified actions we get $O(2^q + m - 1)$ transitions in the resulting automaton.

Analysis of DS-qualified actions

Let H be a hybrid automaton with n locations, m *action_sync* labeled transitions, k variables and let q be the number of **DS**-qualified actions.
While the number of states will once again not change, the number of variables will be increased by $k + 2q$. The reason for this is that we need in addition to the variable, which stores the timestamp one *setter* variable, which is used, if the action is set. As a consequence the number of edges increases in dependency of both setter and timer variables.
To set the action we must duplicate the *action_sync* labeled self loops of the location, where the corresponding step in the SFC contains the **DS**-qualified action. As before we can estimate the worst case by adding all **DS**-qualified actions to one step in the SFC, what means that they will be assigned to only one location in the hybrid automaton. By adding the transitions, which execute the *setter* control action we get $O(2^q + m - 1)$ transitions. But now we must apply the *setter* transformation and get $O(2^{2q} + m2^q - 2^q)$.

Analysis of SL-qualified actions

Let H be a hybrid automaton with n locations, m *action_sync* labeled transitions, k variables and let q be the number of **SL**-qualified actions.
The transformation of **SL**-qualified actions is comparable with the transformation of *setter* actions. The number of states remains unchanged, but the number of variables will be increased by q. To transform the transitions we must duplicate for each **SL**-qualified action every already existing *action_sync* transition. So that the first transition can be taken if the limit time is elapsed and the second one otherwise. We get $O(m2^q)$ transitions in the resulting automaton.

Analysis of SD-qualified actions

Let H be a hybrid automaton with n locations, m *action_sync* labeled transitions, k variables and let q be the number of **SD**-qualified actions.
Analogous to the transformation of **SL**-qualified actions, we must add q variables, while the number of locations remains constant. Also in this case we must duplicate each existing *action_sync* transition and as in the analysis of **SL**-qualified actions we need the first transition to execute the timed action, if the delay time elapses and the second transition otherwise. So we get again $O(m2^q)$ transitions in the resulting automaton.

Analysis results

We give an overview of the analysis results. Let q be the number of **L**-, **D**-, **DS**-, **SL**-and **SD**-qualified actions.

	Locs	Var	Trans
HA	n	k	m
L	n	$k+q$	$O(2^q + m - 1)$
D	n	$k+q$	$O(2^q + m - 1)$
DS	n	$k+2q$	$O(2^{2q} + m2^q - 2^q)$
SL	n	$k+q$	$O(m2^q)$
SD	n	$k+q$	$O(m2^q)$

Also in the case of timed action qualifiers we get exponential growth in the number of transitions. The qualifiers **L** and **D** extend the set of transitions rather local, that means they change only the transitions of their corresponding locations. For stored qualifiers **SL** and **SD** we nearly get the same results as for the *setter* qualifier transformation. The reason for this is that we use the same concept with adapted transition guards and control actions. A special case are **DS**-qualified actions. In this case the setter part must be considered in particular.

Chapter 5
Implementation

Besides the conception of the transformation of memory- and time-related SFCs into hybrid automata, regarded in Chapter 3 and Chapter 4 the integration of the transformation process into the SFC Verification Tool[1] and detailed tests of the implementation are a further part of this thesis.
In this chapter we give a brief introduction to the SFC Verification Tool, describing its capabilities and the verification process. We further give a short introduction into the code structure and the ideas and concepts of the program architecture. In a second part we consider the implementation of the transformation of timed and memory-related SFCs. Finally we give some experimental results of the SFC Verification Tool.

5.1 The SFC Verification Tool

The SFC Verification Tool allows the automatic verification of SFCs under consideration of a specific plant setup. Thus engineers can check their process specifications in a modeled hardware context to ensure that safety conditions are satisfied. In this way crucial errors and malfunctions can be located at the concept phase even before the SFC is used in the plant.
In the following we explain the verification process of the SFC Verification Tool and introduce the program architecture.

5.1.1 The verification process

The verification process is based on the automatic transformation of an SFC into a hybrid automaton. On such a hybrid automaton we can perform a reachability analysis to test safety conditions. Therefore we can divide the verification process into two parts (Figure 5.1). The first part (*Model creation*) transforms a set of SFCs into a set of hybrid automata under consideration of specific plant behavior and the second part (*CEGAR (Counter Example Guided Abstraction Refinement) based verification*) performs a CEGAR-based analysis.
In the *Model creation* process we transform a given set of SFCs into a set of hybrid automata as described in the foregoing chapters. We get an automaton representing the SFC and optionally the dynamic plant specifications. The latter is given as a set of conditional ODE systems describing plant properties as, e.g., tank sizes, pump specifications, etc. The hybrid automata are committed to the second process - the *CEGAR based verification.*
In the *CEGAR based verification* process a reachability analysis on the hybrid automata is performed by the external tool *SpaceEx*[2] that checks, whether forbidden states (*Safety property*) are reachable. The Safety property is given as a set of forbidden states. Since reachability analysis on hybrid automata is undecidable [HK95] the analysis yields an over approximation of the reachable states. Moreover the class of hybrid automata is restricted to the set of automata with linear (PHAVer scenario) or piecewise affine dynamics (LGG support

[1]developed by Johanna Nellen at the chair of computer science 2: *Theory of Hybrid Systems* - RWTH Aachen University
[2]http://www.spaceex.imag.fr

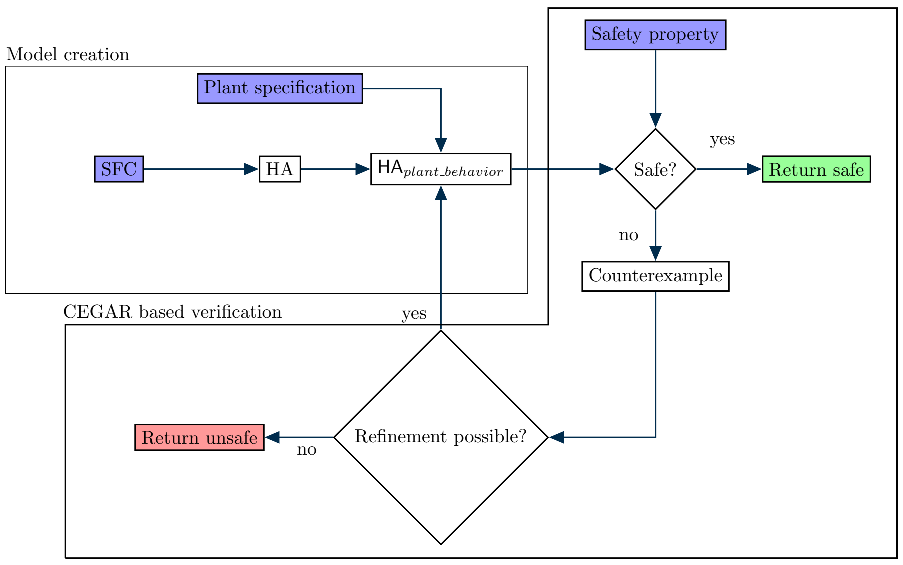


Figure 5.1: CEGAR based refinement.

Function scenario).
Due to the flowpipe computation we have to transform all transition guards and invariants into linear convex equations which again leads to an over approximation [Fre10].
If no forbidden state is reachable the SFC under consideration of its dynamic plant context is safe and we stop the verification process. Otherwise the model is refined by mapping dynamic plant information to locations of the hybrid automata. We can apply different heuristics for the refinement that analyses the analysis results and the current refinement state. In an iterative process, we analyses the refined model until the model is safe or fully refined.
Finally we know either the SFC is safe or the SFC might be not safe (due to the over approximation) under consideration of its plant context.

5.1.2 Program architecture

The architecture of the SFC Verification Tool can be divided into three coarse parts. First the SFC Verification Tool consists of classes, which read the input data from files and store them into data structures. We have further classes, which transform the parsed input data into a data structure for the hybrid automata and classes, which translate the hybrid automata into a tool-specific output format, e.g., a SpaceEx model file. By extending the output writer of the SFC Verification Tool it is easily possible to add support for other analysis tools, like, e.g., Flow*[3].
Because this bachelor thesis covers the transformation of a memory- and time-related SFC into hybrid automata, we focus on the transformation from an SFC into hybrid automata.
The data structure for hybrid automata consists of a class representing the hybrid automata with fields containing the variables, locations and transitions,

[3] http://systems.cs.colorado.edu/research/cyberphysical/taylormodels/

which are part of the automaton. To enable the transformation of memory-and time-related SFCs as described in Section 3 and Section 4, we must modify the automata as follows: 1. Steps and 2. variables in the SFC must be transformed into locations and variables of a hybrid automaton. 3. Transitions must be adopted from the SFC and translated into those of a hybrid automaton. Thereby self loops will be added for every location and actions of the SFC will be encoded in the transitions of the hybrid automaton. Our changes cover only the second and the third transformation processes. We adapt the guards of existing transitions, add new variables and additional transitions. Since the transformation of memory- and time-related SFC does not need any adaption of the step transformation we omit it at this point.
The modifications cover those classes, which translate the variables and transitions of an (H)SFC into a hybrid automaton. To explain the modifications we give a more detailed overview of the transformation-classes in the following.
The class *VariablesHA* (Figure 5.2) contains two fields: A list of variables and the SFC, which must be transformed into a hybrid automaton. If the method *createVariables()* is called the necessary variables are extracted from the SFC and stored in the list of variables.

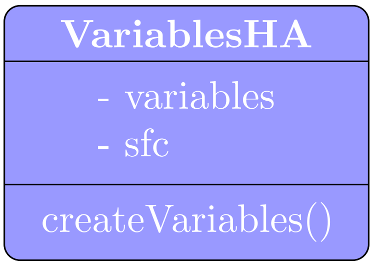
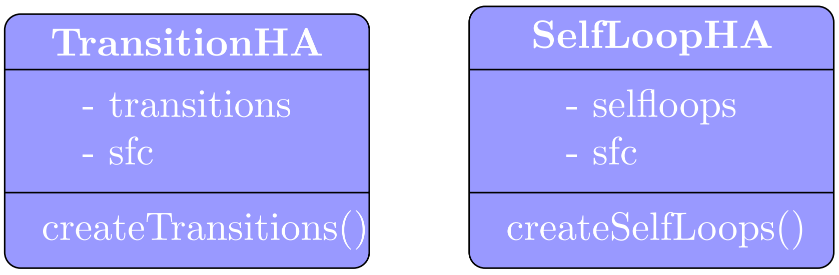


Figure 5.2: Class VariablesHA.

In the SFC Verification Tool we distinguish between self loops and transitions between two different locations. Therefore we have separate classes for the transformation of transitions *TransitionsHA* (Figure 5.3a) and the creation of self loops *SelfLoopHA* (Figure 5.3b).



(a) Class TransitionsHA. (b) Class SelfLoopHA.

As in the transformation of variables we extract the transitions from the SFC using the function *createTransitions()* and store them in the list *transitions*. Analogously, we create self loops with the method *createSelfLoop()* for each existing step in the SFC. Moreover the self loops, which are part of the SFC are stored in the list *selfloops*.

This coarse overview of architecture and the class design of the SFC Verification Tool allows us to explain the concrete adaption and extensions of the source code in the next chapter. Although this is not a complete overview it faces all important changes.

5.2 The implementation

The implementation comprises the adaption and extension of the classes described in Section 5.1 and the creation of a new class *SetterTimedTransHA* (Figure 5.3). This class charges all *setter* and *timed* variables and provides methods, which are used in the classes SelfLoopAutomatonFromPOUCreator, TransitionsAutomatonFromPOUCreator and VariablesAutomatonFromPOUCreator that transform the self loops, the transitions and the variables of an SFC into corresponding elements in the hybrid automaton.

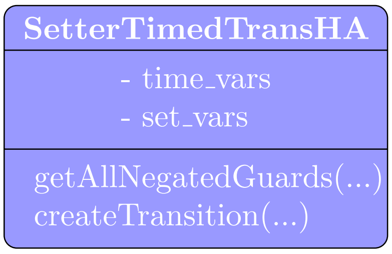


Figure 5.3: Class SetterTimedTransHA.

The method *getAllNegatedGuards()* returns the disjunction of all negated setter guards (Definition 16) and all negated timed guards (Section 4.6.3). We use this method in the following to add the result as a new conjunctive term to the guards of the existing transitions. We need the negated variables, because we want to take the existing transitions, if no setted or timed action must be executed. If one ore more timed and setter actions must be executed those transitions have a higher priority, which contain the corresponding guards (not negated). The method *createTransition()* adds one transition for every combination of non-negated (timed or setter) guard (Definition 19 and Section 4.6.3) to the hybrid automaton. These two methods work on the variables provided by the fields *time_vars* and *set_vars*. We will now consider the transformation of the variables and transitions in detail.

5.2.1 Additional variables

In the class *VariablesHA* we extend the method *createVariables()* and add for every timed- and set-qualified action of the SFC one further variable to the hybrid automaton and one to the *SetterTimedTransHA* list. It is important that the variables are created before the transitions are transformed, because the *SetterTimedTransHA* class needs those variables to construct the correct guards and assignments.

5.2.2 Transitions

We extend the method *createTransition()* of the class *TransitionHA* in two steps. The first step extends the guard of the currently created transition to all negated setter guards and all negated timed guards. Therefore we call the method *getAllNegatedGuard()* of the class *SetterTimedTransHA*. In the second step we create one additional transition for every combination of true valuated timed and setter guard by calling the method *createTransition()* of the class *SetterTimedTransHA*.
We do this for the transitions leading from one location to another one as well as for self loops. We further equip the assignments of the created and adapted transitions with assignments, which are used to (re)set the timer and setter variables.

5.2.3 Initial valuation

If there exists a timed- or set-qualified action in an initial step of an SFC, we cannot activate it using assignments on the incoming transitions of a step. Instead we must adapt the initial valuation of variables. We check in the class *VariablesHA* for each variable if it belongs to an action, which is active in the initial step. If this is the case we valuate the variable respectively.

5.3 Experimental results

The benchmarks considered in the following are memory- and time-related SFCs derived from the plant model described in Section 2.1.
We provide four conditional ODEs (Figure 5.4) describing the dynamic behavior of the plant model. The first conditional ODE system describes the situation where the waterlevels of the tanks T_1 and T_2 will not change. This is the case if tank T_2 is full, tank T_1 is empty or the mixer M_1 or the pump P_1 are not running.
The second conditional ODE system describes the situation where the waterlevel of tank T_1 decreases by 1 per time unit and the waterlevel of tank T_2 increases by 1 per time unit. Therefore the tank T_1 must not be empty ($waterlevel1 \geq min1$), tank T_2 must not be completely filled ($waterlevel2 \leq max2$) and the mixer M_1 and the pump P_1 must be running.
We create SFCs, which define the processes on the plant model. These SFCs provide **S**, **R**, **L** and **D** qualified actions and allow the analysis of memory- and time-related qualifier with the SFC Verification Tool. In the plant model we use these actions to activate and deactivate the mixer M_1.
The aim is to allow only the activation of the pump if the mixer is running. That means we define a forbidden state $chkb\_mixer == 0 \wedge chkb\_p1 == 0$ and check with the SFC Verification Tool, whether this state can be reached.
In the following we take a closer look on the benchmark results obtained from the SFC Verification Tool and give a detailed overview at the end of this chapter.

5.3.1 Mixer Simple

In the *Mixer Simple* benchmark we check the transformation of **S**- and **R**-qualified actions. That means we check the functionality of the in this thesis introduced **S** and **R** qualifier. The SFC that we use for this benchmark is

```
<?xml version="1.0" encoding="utf-8"?>
<condODEsys>
  <condODE>
    <cond>waterlevel1 <= min 1 OR waterlevel2 >= max2 OR NOT chkb_p1 OR
NOT chkb_mixer</cond>
    <equation>waterlevel2' == 0</equation>
    <equation>waterlevel1' == 0</equation>
  </condODE>
  <condODE>
    <cond>waterlevel1 >= min1 AND waterlevel2 <= max2 AND chkb_p1 AND
chkb_mixer</cond>
    <equation>waterlevel2' == 1</equation>
    <equation>waterlevel1' == -1</equation>
  </condODE>
 </condODEsys>
```

Figure 5.4: Conditional ODE system of the *Mixer Simple* benchmark.

described in Section 3.1.
The SFC Verification Tool needs four iterations to proof the correctness of the model (Table 5.7). The SFC Verification Tool examines that the forbidden state is reached in the discrete model. In the first refinement step the first conditional ODE system of Figure 5.4 is attached to the step *Empty*. In the second iteration the step *RunPump* is refined by the second conditional ODE system of Figure 5.4. The third iteration refines the step *Start* with the first conditional ODE system of Figure 5.4. At this point the forbidden state can be still reached. In the last iteration a complete refinement is done and as a result no forbidden state can be reached and the model is correct.

5.3.2 Mixer Simple Timed D

The SFC of the *Mixer Simple Timed Delayed* benchmark (Figure 5.5) is similar to the *Mixer Simple* benchmark. We test with this benchmark the functionality of the **D** qualifier. The only difference concerns the mixer M_1. We start it delayed for 100 milliseconds and do not stop it in the last step *Empty*. The SFC Verification Tool needs four iteration for analysis (Table 5.7). In the first iteration the step *Empty* is refined by the first conditional ODE system of Figure 5.4. The same conditional ODE system is attached to step *RunPump* in the second iteration and to step *Start* in the third iteration. As in the analysis of the *Simple Mixer* benchmark a full refinement is done in the fourth step, that means every conditional ODE systems will be attached to every step. This results in a correct model.

5.3.3 Mixer Simple Timed L

The SFC in Figure 5.6 is also derived from the SFC described in Section 3.1. In this benchmark we test the **L** qualifier, which is introduced in this thesis. We start the mixer M_1 for a limited time of 100 milliseconds. Also for this benchmark the SFC Verification Tool needs four refinement steps to proof the

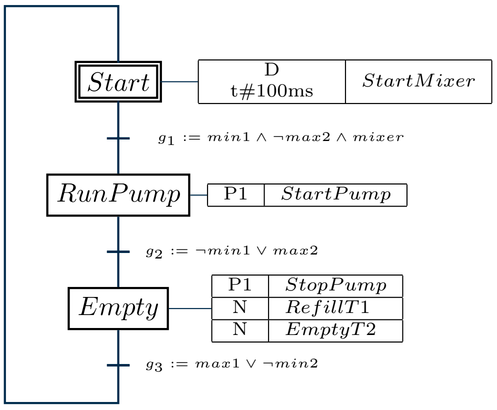

Start
D
t#100ms
StartMixer
$g_1 := min1 \wedge \neg max2 \wedge mixer$
RunPump
P1
StartPump
$g_2 := \neg min1 \vee max2$
Empty
P1
StopPump
N
RefillT1
N
EmptyT2
$g_3 := max1 \vee \neg min2$

Figure 5.5: SFC of the *Simple Mixer Timed Delayed* benchmark.

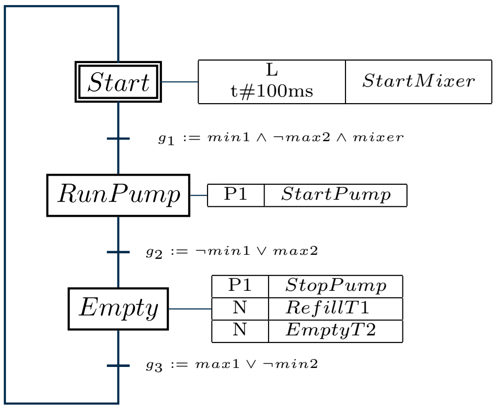

Start
L
t#100ms
StartMixer
$g_1 := min1 \wedge \neg max2 \wedge mixer$
RunPump
P1
StartPump
$g_2 := \neg min1 \vee max2$
Empty
P1
StopPump
N
RefillT1
N
EmptyT2
$g_3 := max1 \vee \neg min2$

Figure 5.6: SFC of the *Simple Mixer Timed Limited* benchmark.

model correctness (Table 5.7). In the first, second and third refinement step the first conditional ODE system of Figure 5.4 is attached successively to the steps *Empty*, *RunPump* and *Start*. Thereafter, in the fourth refinement step, a full refinement is applied on the model, that means every conditional ODE system is attached to every step of the SFC. Finally the model is stated to be correct.
The idea behind the *Mixer Simple Timed L* benchmark was the creation of an incorrect benchmark, which tests the limited execution of the mixer M_1. The mixer M_1 should be stopped after the limit time, while the pump P_1 is still activated, which results in a forbidden state. Actually, this forbidden state is not reached and the model is correct. The reason for that is that the *StartMixer* action starts the mixer M_1 and lets it run until the mixer is explicitly stopped by a stop action. So if a limited action sets a value in a variable, e.g., a flag, the variable is set in each time unit, where the action is active, but it is not defined that the flag will be resetted after the time limit elapses.
To correct this benchmark we could increase the number of rotation for a limited time and if the time elapse the number of rotation remains constant.

Table 5.7 provides an overview of the benchmark results. Besides the infor-

Benchmark	# ref.	Result	Exec. time	Safety property
Mixer Simple	4	Correct	5:15min	chkb_mixer = 0 ∧ chkb_p1 = 1
Mixer Simple Timed D	4	Correct	6:30min	chkb_mixer = 0 ∧ chkb_p1 = 1
Mixer Simple Timed L	4	Correct	7:02min	chkb_mixer = 0 ∧ chkb_p1 = 1

Figure 5.7: Overview of the experimental results.

Benchmark	SFC Size	HA Size in refinements (Var/Step/Trans)				
	Var/Loc/Trans	0	1	2	3	4
Mixer Simple	12/3/23	15/3/23	15/7/67	15/11/167	15/15/275	15/33/627
Mixer Simple Timed D	12/3/3	15/3/19	15/7/43	15/11/95	15/15/175	15/36/612
Mixer Simple Timed L	12/3/3	15/3/19	15/7/43	15/11/95	15/15/175	15/36/612

Figure 5.8: Overview of the SFC and HA sizes.

mations already discussed in the sections before we get an impression of the runtime of the different benchmarks. Although the *Mixer Simple* benchmark is the greatest one of the three benchmark regarding variables, locations and transitions (Table 5.8) it needs the shortest time for the analysis. A reason can be that we do not need to consider a timer in the verification process, that means we have fewer continuous variables in the model. The *Mixer Simple* benchmark model contains (considerable) more transitions than the *Mixer Simple Timed D* or *Mixer Simple Timed L* benchmark. If we remember the transformation process we know that we must adapt the whole automaton if we transform **S** and **R** qualified actions, while we must only adapt the incoming-,outgoing- and self loop-transitions of locations, where the corresponding step contains **L**, **D** qualified actions. In both cases, for **L** and **D** qualified actions, we duplicate the transitions of the corresponding locations. The substantial difference affects the guard, which is added to the transitions.
But the *Mixer Simple Timed D* and the *Mixer Simple Timed L* benchmarks are nearly of the same size and need different runtime for the analysis. An explanation for that must be the different guard evaluations, because the two benchmarks are very similar.

Chapter 6

Practical example

In the following we will demonstrate the application of our definitions and concepts. We introduce a complex example and apply the different transformations on it. So that we have finally a good impression of the transformation process and the increase of the number of transitions in the resulting hybrid automaton.

Mixer model

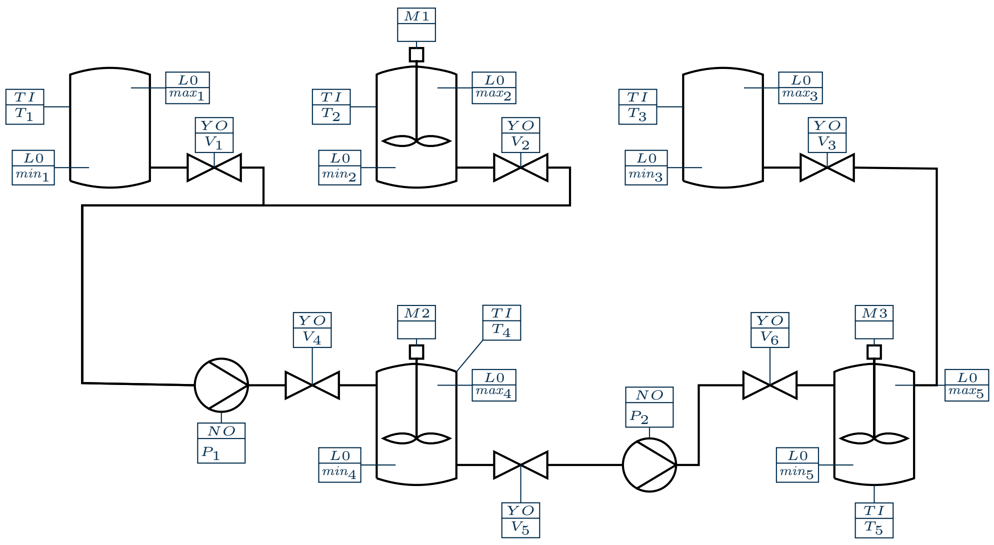


Figure 6.1: Mixer Model.

The complex mixer model (Figure 6.1) consists of five tanks $T_1, ..., T_5$, six valves $V_1, ..., V_6$, three mixer M_1, M_2, M_3 and two pumps P_1, P_2. The tanks are equipped with sensors $min_i, max_i, i \in [5]$[1], which announce if a tank is empty or completely filled. The tanks T_1 and T_2 are connected to tank T_4 by pipes. All tanks are cylindrical with the same diameter, but different heights. A pump P_1 and valves V_1, V_2, V_4 allows to pump liquid through pipes from one tank into another. The tanks T_3 and T_4 are connected to tank T_5. A further pump P_2 and valves V_5, V_6 allows to decant liquid from tank T_4 to tank T_5. Tank T_3 can decant liquid to tank T_5 using gravity by opening valve V_3. Three mixers M_1, M_2, M_3 are installed in the tanks T_2, T_4 and T_5.
We introduce variables to access the configurations of the different components. $min_i, max_i \in [5]$ are true, if the waterlevel of the corresponding tank covers the sensor. The variables P_1, P_2 and $mixer_i, i \in [3]$ are true if the pump or the mixer are running. Otherwise false. For the valves, we have variables $Valve_i, i \in [6]$, which are true, if the corresponding valve is opened and false if it is closed. We further introduce variables $h_i, i \in [5]$, which contains the waterlevels of the corresponding tanks. We will use these variables, when we introduce the

[1] $[i] := \{1, ..., i\} \subseteq \mathbb{N}$

conditional ODE systems later on.
Following the simple mixer model, we give a Sequential Function Chart, which describes the execution process (Figure 6.2).
The execution starts by activating the mixer M_2 of tank T_4. This action will be stored and reset in the last step *Ready*, so that the mixer M_2 runs in all steps between. If the mixer M_2 is running, the tank T_1 is completely filled and the tank T_4 is empty we take the first transition and enter the next step *Drain_Tank1*. We open the valves V_1, V_4 and start the pump P_1 to decant the liquid of tank T_1 into tank T_4. Thereby we start the mixer M_1 of tank T_2, which is needed in the next step. The reason for that is that the mixer needs some time to reach its working speed. We limit the execution to one hour, because the mixer is very sensitive and we want to avoid overheating. If the tank T_1 is drained, the tank T_2 is filled and the mixers M_1, M_2 are running we can enter the next step. Thereby we execute the **P0**-qualified actions of the step *Drain_Tank1*, which are closing the valve V_1 and halting the pump P_1.
The next step *Drain_Tank2* will start the decantation of tank T_2. Therefore the valve V_2 will be opened and the pump P_1 started. We also start the mixer M_3 with a delay of 30 minutes. So the mixer has time to reach its working speed but runs not the whole activation time of step *Drain_Tank2*. We leave the step if tank T_2 is empty, the tanks T_3 and T_4 are filled, the tank T_5 is not filled and all pumps are running. Thereby we stop the pump P_1 and close the valves V_2 and V_4.
The third step *Drain_Tank3* will drain the content of tank T_3 and T_4 into tank T_5. After deactivating the mixer M_1, we open the valves V_3, V_4, V_5 and start the pump P_2. The liquid will be mixed in tank T_5. We change over to the *Ready* step, if tank T_5 in filled and tank T_4 is empty. Thereby we close the open valves V_3, V_4, V_5 and stop the pump P_2. At the end we reset the running mixers $M2$ and $M3$.

We introduce some conditional ODEs, which represents the dynamic behavior of the application context.

$$\begin{aligned}
P_1 \wedge mixer_2 \wedge Valve_1 \wedge Valve_4 &:\ h_4' = c_1,\ h_1' = -c_1 \\
P_1 \wedge mixer_1 \wedge mixer_2 \wedge Valve_2 \wedge Valve_4 &:\ h_4' = c_2,\ h_2' = -c_2 \\
P_2 \wedge mixer_3 \wedge Valve_3 \wedge Valve_5 \wedge Valve_6 &:\ h_5' = c_3 + c_4,\ h_3' = -c_3,\ h_4' = -c_4
\end{aligned}$$

The first conditional ODE can be described as follows: If pump P_1 and mixer M_2 are running and the valves V_1, V_4 are opened the waterlevel h_4 of tank T_4 will increase by a constant value c_1 per time unit, while the waterlevel h_1 of tank T_1 will decrease for the same value.
The second conditional ODE is similar to the first one. In this case we decant tank T_2 into tank T_4, if the pump P_1 and the mixer M_1 are running and the valves V_2, V_4 are open. The waterlevel h_4 of tank T_4 will increase by a constant value c_2 per time unit and the waterlevel h_2 of tank T_2 sinks for the same value per time unit.
The last conditional ODE is more interesting. If the mixer M_3 and the pump P_2 are running and the valves V_3, V_5, V_6 are open the waterlevel h_5 of tank T_5 will increase by $c_4 + c_3$, which are the constant values subtracted from the waterlevels h_3, h_4 of the tanks T_3, T_4.
We will abbreviate the conditional odes by $cond_i : ode_i$ where $i \in [3]$ represents

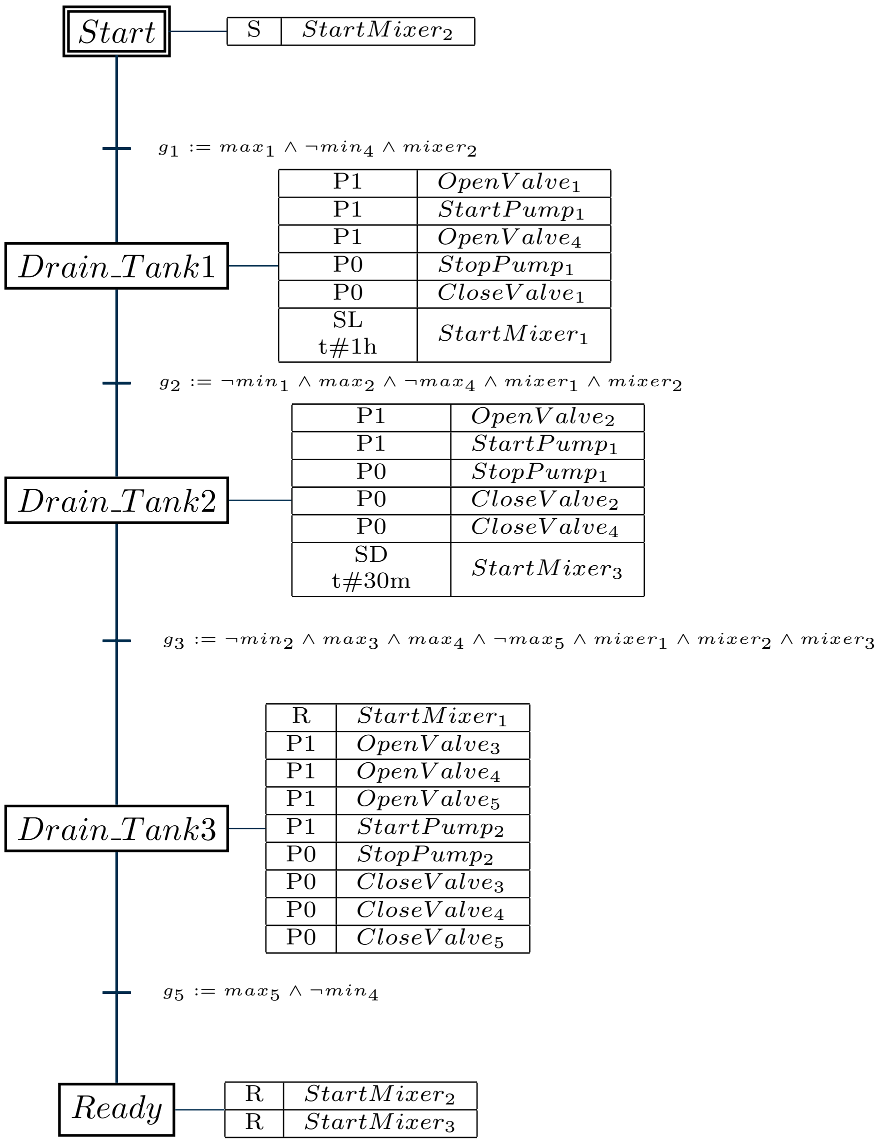


Figure 6.2: Mixer model SFC.

the first, second and third introduced conditional ODE system.

We will now assign the conditional ODEs to steps of the SFC (Figure 6.2) to get a HSFC (Figure 6.3).

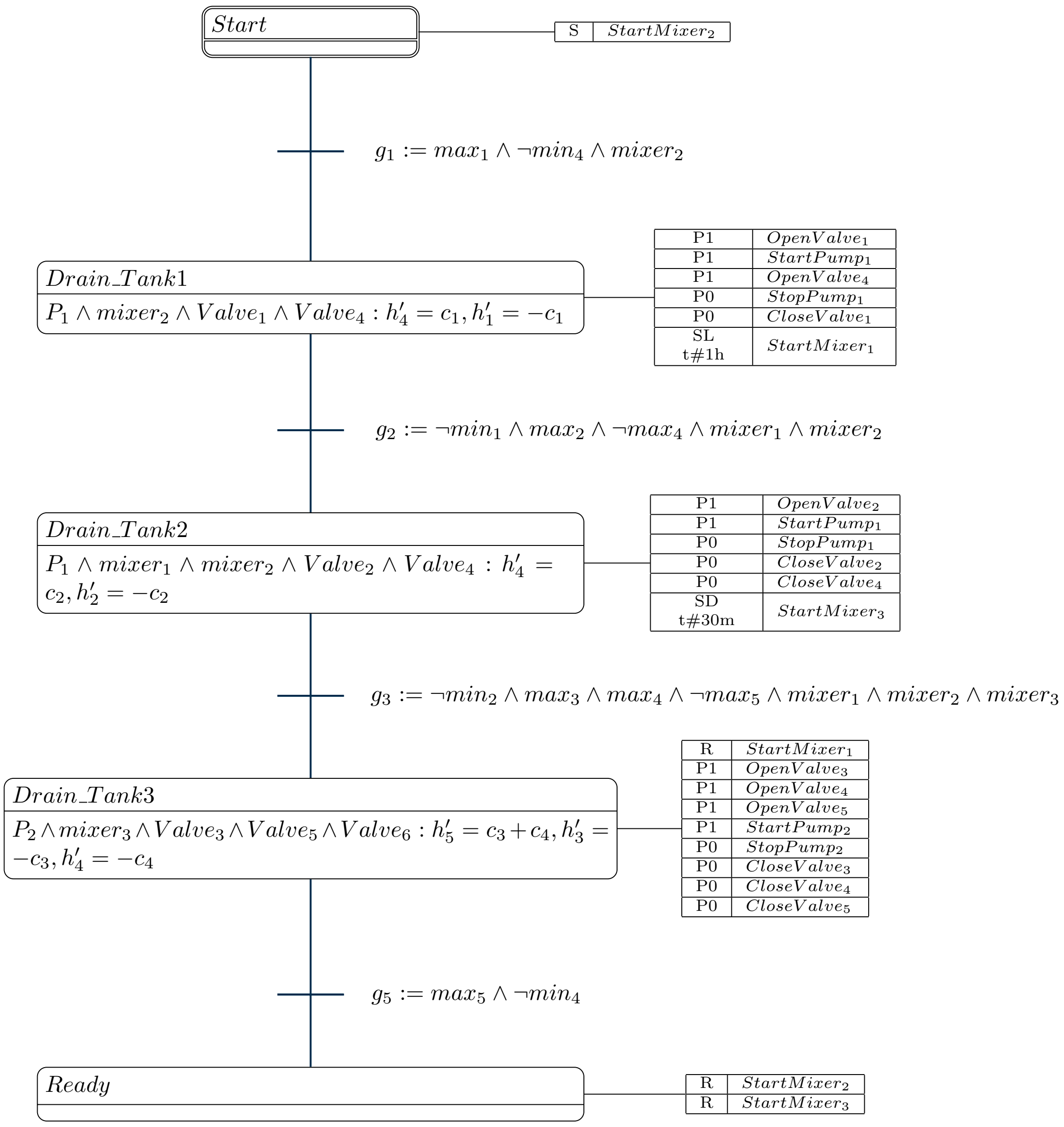

$Start$
S $StartMixer_2$
$g_1 := max_1 \wedge \neg min_4 \wedge mixer_2$
$Drain\_Tank1$
$P_1 \wedge mixer_2 \wedge Valve_1 \wedge Valve_4 : h'_4 = c_1, h'_1 = -c_1$
P1 $OpenValve_1$
P1 $StartPump_1$
P1 $OpenValve_4$
P0 $StopPump_1$
P0 $CloseValve_1$
SL t#1h $StartMixer_1$
$g_2 := \neg min_1 \wedge max_2 \wedge \neg max_4 \wedge mixer_1 \wedge mixer_2$
$Drain\_Tank2$
$P_1 \wedge mixer_1 \wedge mixer_2 \wedge Valve_2 \wedge Valve_4 : h'_4 = c_2, h'_2 = -c_2$
P1 $OpenValve_2$
P1 $StartPump_1$
P0 $StopPump_1$
P0 $CloseValve_2$
P0 $CloseValve_4$
SD t#30m $StartMixer_3$
$g_3 := \neg min_2 \wedge max_3 \wedge max_4 \wedge \neg max_5 \wedge mixer_1 \wedge mixer_2 \wedge mixer_3$
$Drain\_Tank3$
$P_2 \wedge mixer_3 \wedge Valve_3 \wedge Valve_5 \wedge Valve_6 : h'_5 = c_3 + c_4, h'_3 = -c_3, h'_4 = -c_4$
R $StartMixer_1$
P1 $OpenValve_3$
P1 $OpenValve_4$
P1 $OpenValve_5$
P1 $StartPump_2$
P0 $StopPump_2$
P0 $CloseValve_3$
P0 $CloseValve_4$
P0 $CloseValve_5$
$g_5 := max_5 \wedge \neg min_4$
$Ready$
R $StartMixer_2$
R $StartMixer_3$

Figure 6.3: Mixer model HSFC.

We provide the transformed HSFC in Figure 6.4. We write $asgj_i, ...asgj_{i+n}$ (asg = action sync guard) for adding n transitions to a step. The *time_sync* transitions are left out to improve readability just as the detailed guards in the transition table. We write e.g. $g_{StartMixer_1}$ for the guard of the action $StartMixer_1$. Since the action is **SL**-qualified with a limit of one hour the guard is $t_{timer} \leq 3600000 + t_{StartMixer_1}$. In the same way we abbreviate the other

transition guards.
If there is no action behind the right arrow $\rightarrow$, nothing will be executed.

If we take a look on the SFC (Figure 6.2) and the resulting hybrid automaton (Figure 6.4), we get a good impression of the dimensions. A relative simple SFC, with only four transitions and five states will be blowed up to $5,062$ transitions and 8 locations and that without counting *read_sync* transitions. We need also a lot of variables to store the *setter* actions and timestamps of timed actions.

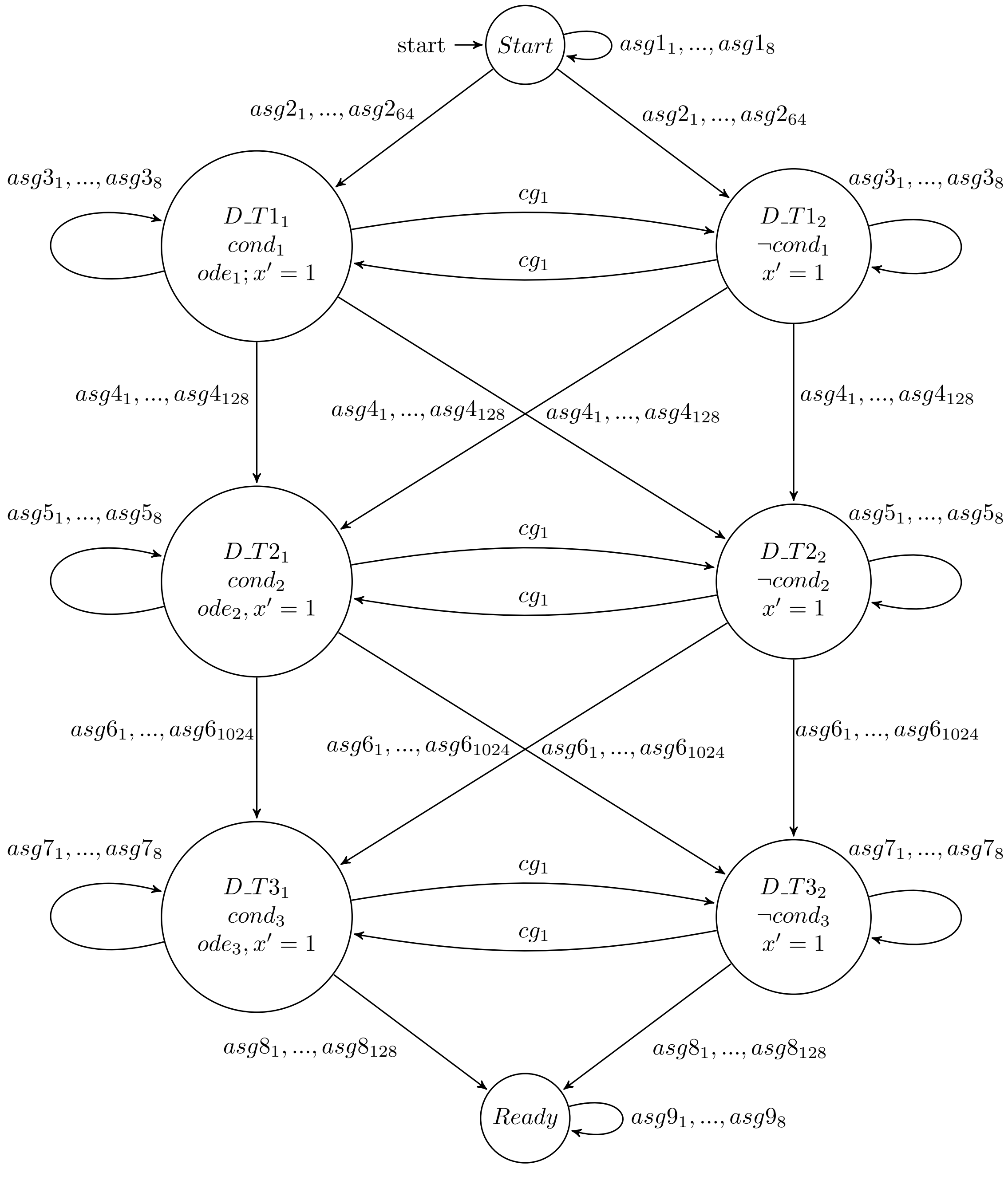


Figure 6.4: Hybrid automaton of the mixer model. (D_T means $Drain\_Tank$).

Abbr.	Label	Guard
cg_1	*copy_trans*	$x \geq \varepsilon \to x := 0$
$asg1_1$	*action_sync*	$\neg g_1 \wedge \neg g_{StartMixer_2} \wedge \neg g_{StartMixer_3} \wedge \neg g_{StartMixer_1} \to$
$asg1_2$	*action_sync*	$\neg g_1 \wedge \neg g_{StartMixer_2} \wedge \neg g_{StartMixer_3} \wedge g_{StartMixer_1} \to StartMixer_1$
$asg1_3$	*action_sync*	$\neg g_1 \wedge \neg g_{StartMixer_2} \wedge g_{StartMixer_3} \wedge \neg g_{StartMixer_1} \to StartMixer_3$
$asg1_4$	*action_sync*	$\neg g_1 \wedge \neg g_{StartMixer_2} \wedge g_{StartMixer_3} \wedge g_{StartMixer_1} \to StartMixer_1; StartMixer_3$
$asg1_5$	*action_sync*	$\neg g_1 \wedge g_{StartMixer_2} \wedge \neg g_{StartMixer_3} \wedge \neg g_{StartMixer_1} \to StartMixer_2$
$asg1_6$	*action_sync*	$\neg g_1 \wedge g_{StartMixer_2} \wedge \neg g_{StartMixer_3} \wedge g_{StartMixer_1} \to StartMixer_1; StartMixer_2$
$asg1_7$	*action_sync*	$\neg g_1 \wedge g_{StartMixer_2} \wedge g_{StartMixer_3} \wedge \neg g_{StartMixer_1} \to StartMixer_2; StartMixer_3$
$asg1_8$	*action_sync*	$\neg g_1 \wedge g_{StartMixer_2} \wedge g_{StartMixer_3} \wedge g_{StartMixer_1} \to StartMixer_1; StartMixer_2; StartMixer_3$
$asg2_1$	*action_sync*	$g_1 \wedge \neg g_{StartMixer_1} \wedge \neg g_{StartMixer_2} \wedge \neg g_{StartMixer_3} \wedge \neg g_{OpenValve_1} \wedge \neg g_{OpenValve_4} \wedge \neg g_{StartPump_1} \to$
$asg2_2$	*action_sync*	$g_1 \wedge \neg g_{StartMixer_1} \wedge \neg g_{StartMixer_2} \wedge \neg g_{StartMixer_3} \wedge \neg g_{OpenValve_1} \wedge \neg g_{OpenValve_4} \wedge g_{StartPump_1} \to StartPump_1$
...	...	...
$asg2_{64}$	*action_sync*	$g_1 \wedge g_{StartMixer_1} \wedge g_{StartMixer_2} \wedge g_{StartMixer_3} \wedge g_{OpenValve_1} \wedge g_{OpenValve_4} \wedge g_{StartPump_1} \to OpenValve_1; OpenValve_4; StartPump_1; StartMixer_1; StartMixer_2; StartMixer_3$
$asg3_1$	*action_sync*	$\neg g_2 \wedge \neg g_{StartMixer_1} \wedge \neg g_{StartMixer_2} \wedge \neg g_{StartMixer_3} \to$
...	...	...
$asg3_8$	*action_sync*	$\neg g_2 \wedge g_{StartMixer_1} \wedge g_{StartMixer_2} \wedge g_{StartMixer_3} \to StartMixer_1; StartMixer_2; StartMixer_3$
$asg4_1$	*action_sync*	$g_2 \wedge \neg g_{StartMixer_1} \wedge \neg g_{StartMixer_2} \wedge \neg g_{StartMixer_3} \wedge \neg g_{OpenValve_2} \wedge \neg g_{CloseValve_1} \wedge \neg g_{StartPump_1} \wedge \neg g_{StopPump_1} \to$
...	...	...
$asg4_{128}$	*action_sync*	$g_2 \wedge g_{StartMixer_1} \wedge g_{StartMixer_2} \wedge g_{StartMixer_3} \wedge g_{OpenValve_2} \wedge g_{CloseValve_1} \wedge g_{StartPump_1} \wedge g_{StopPump_1} \to CloseValve_1; OpenValve_2; StopPump_1; StartPump_1; StartMixer_1; StartMixer_2; StartMixer_3$
$asg5_1$	*action_sync*	$\neg g_3 \wedge \neg g_{StartMixer_1} \wedge \neg g_{StartMixer_2} \wedge \neg g_{StartMixer_3} \wedge \neg g_{OpenValve_2} \wedge \neg g_{StartPump_1} \to$
...	...	...
$asg5_8$	*action_sync*	$\neg g_3 \wedge g_{StartMixer_1} \wedge g_{StartMixer_2} \wedge g_{StartMixer_3} \wedge \to StartMixer_1; StartMixer_2; StartMixer_3$

$asg6_1$	*action_sync*	$g_3 \wedge \neg g_{StartMixer_1} \wedge \neg g_{StartMixer_2} \wedge \neg g_{StartMixer_3} \wedge \neg g_{StopPump_1} \wedge \neg g_{CloseValve_2} \wedge \neg g_{CloseValve_4} \wedge \neg g_{OpenValve_3} \wedge \neg g_{OpenValve_4} \wedge \neg g_{OpenValve_5} \wedge \neg g_{StartPump_2} \rightarrow a^{reset}_{t_{StartMixer_1}}$
...	...	...
$asg6_{1024}$	*action_sync*	$g_3 \wedge g_{StartMixer_1} \wedge g_{StartMixer_2} \wedge g_{StartMixer_3} \wedge g_{StopPump_1} \wedge g_{CloseValve_2} \wedge g_{CloseValve_4} \wedge g_{OpenValve_3} \wedge g_{OpenValve_4} \wedge g_{OpenValve_5} \wedge g_{StartPump_2} \rightarrow a^{reset}_{t_{StartMixer_1}}; CloseValve_2; CloseValve_4; StopPump_1; OpenValve_3; OpenValve_4; OpenValve_5; StartPump_2; StartMixer_1; StartMixer_2; StartMixer_3$
$asg7_1$	*action_sync*	$\neg g_4 \wedge \neg g_{StartMixer_1} \wedge \neg g_{StartMixer_2} \wedge \neg g_{StartMixer_3} \rightarrow$
...	...	...
$asg7_8$	*action_sync*	$\neg g_4 \wedge g_{StartMixer_1} \wedge g_{StartMixer_2} \wedge g_{StartMixer_3} \wedge \rightarrow StartMixer_1; StartMixer_2; StartMixer_3$
$asg8_1$	*action_sync*	$g_4 \wedge \neg g_{StartMixer_1} \wedge \neg g_{StartMixer_2} \wedge \neg g_{StartMixer_3} \wedge \neg g_{CloseValve_3} \wedge \neg g_{CloseValve_4} \wedge \neg g_{CloseValve_5} \wedge \neg g_{StopPump_2} \rightarrow a^{reset}_{\xi_{StartMixer_2}}; a^{reset}_{t_{StartMixer_3}}$
...	...	...
$asg8_{128}$	*action_sync*	$g_4 \wedge g_{StartMixer_1} \wedge g_{StartMixer_2} \wedge g_{StartMixer_3} \wedge g_{CloseValve_3} \wedge g_{CloseValve_4} \wedge g_{CloseValve_5} \wedge g_{StopPump_2} \rightarrow a^{reset}_{\xi_{StartMixer_2}}; a^{reset}_{t_{StartMixer_3}}; CloseValve_3; CloseValve_4; CloseValve_5; StopPump_2; StartMixer_1; StartMixer_2; StartMixer_3$
$asg9_1$	*action_sync*	$\neg g_{StartMixer_1} \wedge \neg g_{StartMixer_2} \wedge \neg g_{StartMixer_3} \rightarrow$
...	...	...
$asg9_8$	*action_sync*	$\neg g_{StartMixer_1} \wedge \neg g_{StartMixer_2} \wedge \neg g_{StartMixer_3} \rightarrow StartMixer_1; StartMixer_2; StartMixer_3$

Figure 6.5: Transition guide.

Chapter 7
Conclusion

We saw in the foregoing sections, how we can extend HSFCs to time and memory related action qualifiers.
In Section 3 we extended the syntax and semantics of HSFCs with respect to set and reset action qualifiers and give instructions for transforming a *setter* HSFC into a hybrid automaton. Thereby we use *setter* variables to indicate whether an action will be executed or not. We notice that we must consider every possible combination of *setter* variables, what means that we add for each possible combination and for each existing *action_sync* transition one further transition to the automaton. We determine that the resulting automaton will have $O((m + n) * 2^{k+u} + n)$ transitions, where n is the number of steps, m the count of transitions and k, u the number of variables in the unaltered SFC. The automaton will increase exponentially with respect to its transitions.
The computation time of the algorithm, which computes all possible combinations of setter variables is a further challenge. We have seen in Section 3.4 that the algorithm (Algorithm 3) needs a time of $O(n2^n)$ for extracting all possible combinations of *setter* variables, where n is the number of *setter* variables. In the worst case we must do this for every step in the SFC. We accept the necessity of minimizing the set of *setter* variables.
In Section 4 we have introduced the syntax and semantics of timed (H)SFCs. We further gave instructions to transform a timed SFC into a hybrid automaton to enable its verification. Thereby we must distinguish between stored timed actions and those, which are only active if their corresponding step is active. For the latter we only add transitions to the locations, where the corresponding step in the SFC contains **L** or **D**-qualified actions. But for the stored timed actions we need to add all possible combinations of *setter* variables to all existing *action_sync* labeled transitions. That means we must always add one transition in which the limited or delayed action will be executed and one, if this is not the case. Thereby we face the same problems as we considered for set and reset qualifiers. The number of transitions rises exponentially in the resulting automaton.
As a result of this thesis, we should minimize the number of *set*, *reset* and stored time-qualified actions, because if we use them we always must consider the case that the corresponding action will not be executed. Finally we duplicate every transition for each stored qualifier. But there are ideas to improve the transformation.
As a future work we could create *setter* variables only for those actions, which will be reset in the SFC. This can be decided by doing a DFS or BFS on the SFC. For the worst case in which all actions will be reset this will change nothing, but in practical usage we can assume that there exists actions, which will not be reset and in the case of exponential space complexity every variable decreases the computation time distinctly.
Another idea is to do a reachability analysis for *set*-qualified actions. If we are able to comprehend the possible activation scenarios, we can replace the *set* by other qualifiers and remove its corresponding reset. That means if e.g. an action is set in a step and will be reset in the next one, and there are no parallel

executed actions (in nested SFCs or parallel branches) we can replace the **S** by an **N** and remove the *reset* in the second step.
These are some ideas, which can improve the time and space complexity of the algorithms and transformation instructions in practical usage. But in the worst case we are neither able to find variables, which were never reset, nor we can determine exact activation and deactivation points for **S**-qualified actions.

Bibliography

[ACH$^+$95] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theor. Comput. Sci.*, 138:3–34, 1995.

[Bau04] N. Bauer. *Formale Analyse von Sequential Function Charts.* PhD thesis, Dortmund University of Technology, 2004.

[BCMP] L. Baresi, S. Carmeli, A. Monti, and M. Pezzè. PLC programming languages: A formal approach. `http://www.google.de/url?sa=t&rct=j&q=plc%20programming%20languages%20a%20formal%20approach&source=web&cd=1&ved=0CDUQFjAA&url=http%3A%2F%2Fhome.dei.polimi.it%2Fbaresi%2Fpapers%2Fanipla2.pdf&ei=cmMyUdDNAsrWtAbvh4C4Dw&usg=AFQjCNEcP1KL5ZHwx7sKYRmc8ZkTc1vllA&bvm=bv.43148975,d.Yms&cad=rja`.

[BHLE04] N. Bauer, R. Huuck, B. Lukoschus, and S. Engell. A Unifying Semantics for Sequential Function Charts. In *SoftSpez Final Report*, volume 3147 of *Lecture Notes in Computer Science*, pages 400–418, 2004.

[Fre10] G. Frehse. An Introduction to SpaceEx v0.8. `http://spaceex.imag.fr/sites/default/files/introduction_to_spaceex_0.pdf`, 2010.

[HK95] T.A. Henzinger and P.W. Kopke. *Undecidability results for hybrid systems.* Technical report (Cornell University. Mathematical Sciences Institute). 1995.

[HKD98] G. Hassapis, I. Kotini, and Z. Doulgeri. Validation of a SFC software specification by using hybrid automata. In *Proc. of INCOM*, pages 65–70, 1998.

[Int03] Int. Electrotechnical Commission. *Programmable Controllers, Part 3: Programming Languages, 61131-3*, 2003.

[NÁ12] J. Nellen and E. Ábrahám. Hybrid Sequential Function Charts. In J. Brandt and K. Schneider, editors, *MBMV*, pages 109–120, 2012.