Towards Automatic Verification of Embedded Control Software

Nanette Bauer
University of Dortmund, Department of Chemical Engineering
Emil-Figge-Str. 70, D-44221 Dortmund, Germany
n.bauer@chemietechnik.uni-dortmund.de

Ralf Huuck
University of Kiel
Institute of Computer Science and Applied Mathematics
Preußerstr. 1–9, D-24105 Kiel, Germany
rhu@informatik.uni-kiel.de

Abstract

The language sequential function charts *(SFC) is a programming and structuring language for programmable logic controllers (PLC). It is defined in the IEC 61131-3 standard and includes various interesting concepts such as parallelism, hierarchy, priorities, and activity manipulation. Although SFCs are perpetually used in the engineering community for programming and the design of embedded control systems, there are hardly any specific verification approaches for them. Existing approaches for Petri Nets, Grafcets, or (UML-)Statecharts do not really apply to SFCs, whose structures are similar, but include distinct features. In this work we present a method to model-check SFCs. This is done by defining a translation of SFCs into the native language of the Cadence Symbolic Model Verifier (CaSMV). This translation is specifically tailored to cover all the concepts of SFCs and can be performed automatically. Moreover, we demonstrate our approach by an application to a control process in chemical engineering.*

1. Introduction

Industrial manufacturing, transportation, chemical engineering and many other applications need increasing numbers of ever new automated machines and systems. They control production lines, transportation units or chemical plants. For a vast part the target system is based on programmable logic controllers. Certainly, their software plays a critical role and its correctness is highly desirable [6].

SFCs are unique among the existing programming languages for PLCs [11], since they define a high level graphical description language which allows parallelism, hierarchy and activity manipulation. Intuitively, SFCs are a particular kind of transition systems where each node is associated with a set of programs, which in turn can be SFCs. These programs can be "switched" on or off in various ways, depending on *qualifiers* associated to a program in a node. This allows, for instance, the activation of a program in node n_1 and its continuous execution until a node n_2 is reached, where it is switched off again. Hence, an agent-like activation and deactivation of programs is supported and opens interesting fields of applications. However, at the same time it is often far from obvious to imagine all the system configurations, i.e., the set of active and non-active programs at a time. Therefore, one goal in this area of research is an automatic analysis of SFCs and the verification of their requirements.

There exist a few approaches to prove the correctness of SFCs. However, previous work [14, 15, 10] does not consider the concepts of hierarchy and priorities or deals with a simplistic concept only [7], which is not well-suited for practical application. Moreover, no one takes the concept of activity manipulation into account. This work intends to remedy the current situation by covering all the major concepts as well as taking care of the distinct nature of PLC execution. The formal verification is done by model checking [8], [19]. In order to do so, we concentrate on a translation of SFCs into the native language of the model checker CaSMV [16] with respect to the aforementioned setting. We consider all the main concepts discussed before and demonstrate our verification method by an application to the control software of a chemical plant.

The remainder of this paper is structured as follows: First, we introduce sequential function charts and briefly present their syntax and semantics in Sect. 2. In Sect. 3 we define a translation from SFC to CaSMV's native code and

illustrate this process in Sect. 4 by a case study. Finally, we draw some conclusions and point out directions for future research in Sect. 5.

2. The SFC Language

Sequential function charts are defined in IEC 61131-3 standard [11] as elements of a graphical programming and structuring language for programmable logic controllers. The SFC definitions are based on the IEC 60848 standard [12], which defines the specification language Grafcet. Grafcet in turn is strongly related to Petri nets [9].

Basically, SFCs are transition systems consisting of *steps* (the locations) and *transitions*. For every SFC there exists exactly one *initial step*. Every transition is labelled by an associated transition condition, called *guard*. Moreover, one or more *actions* may be associated to each step. Actions are again SFCs or programs in one of the other programming languages proposed by the standard. Furthermore actions simply may be given by a boolean variable. Since the actions associated to steps can be SFCs themselves, a concept of hierarchy is provided. An example of a sequential function chart is depicted in Fig. 1.

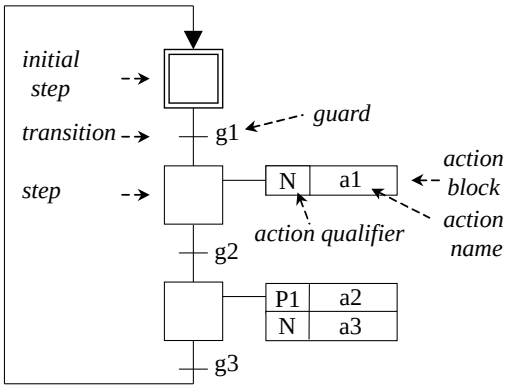

Figure 1. Elements of SFCs

The *action blocks* shown in Fig. 1 are a graphical means to associate actions to steps. An action block consists of an *action qualifier*, which can be used to specify the activity of the respective action, and the *action name*. Concerning the action qualifiers defined in the standard, we concentrate on those without an associated duration of time. These are the following:

- `N` – ***N**on-stored*
- `R` – *overriding **R**eset*
- `S` – ***S**et or stored*
- `P1` – ***P**ulse – rising edge*
- `P0` – ***P**ulse – falling edge*

Intuitively, the non-stored actions are always active while control resides in the corresponding step, i.e., the step is active. In contrast the stored actions keep on being active even outside their step of activation until the corresponding reset action is called. The actions with the pulse qualifier are performed only once when entering (`P1`) or exiting (`P0`) a step, i.e., when the step gets activated or deactivated. If the action is a boolean variable, the variable is true if the action is active and false otherwise.

An SFC does not necessarily have to be a single sequence of steps and transitions. It is possible to have *alternative choice* which means that more than one transition branches from one step as shown in (2) of Fig. 1, and *parallel* or *simultaneous* sequences, which denotes that one transition synchronizes the evolution of two or more sequences, as shown in (1). Moreover, the standard allows *loops* to other steps, which can be interpreted as a particular kind of alternative branching.

These basic transition types can be combined into more complex transition structures. However, there are various combinations which do not seem to make sense, e. g., as shown in Fig. 2 (3), the loop out of the parallel branch. In the following we assume all SFCs to be well-formed, i.e., free of such constructions.

An additional feature of SFCs is the possibility of explicitly assigning priorities to alternative branches. This means, when firing transitions the one which has the highest priority among the enabled ones will be taken. If there are no priorities given explicitly there still is the implicit rule that the transitions are ordered "from left to right" in decreasing priority.

Formally, we define the syntax of an SFC as follows:

Definition 1 (SFC) *A* sequential function chart *(SFC) is a 7-tuple* $\mathcal{S} = (S, A, s_0, T, block, \sqsubset, \prec)$*, where*

- S *is a finite set of* steps,
- A *is a finite set of* action blocks, *which might be SFCs,*
- s_0 *is an* initial step *in* S*,*
- $T \subseteq (2^S \setminus \{\emptyset\}) \times G \times (2^S \setminus \{\emptyset\})$ *is a finite set of* transitions,
- $block : S \to 2^B$ *is an* action labelling function *which assigns a set of action blocks to each step,*
- $\sqsubset \subseteq A \times A$ *is an irreflexive partial order on actions, which might be empty,*
- $\prec \subseteq T \times T$ *is an irreflexive partial order on transitions, which might be empty.*

The order on transitions ensures to cover priorities on conflicting transitions, i.e., transitions with non disjoint

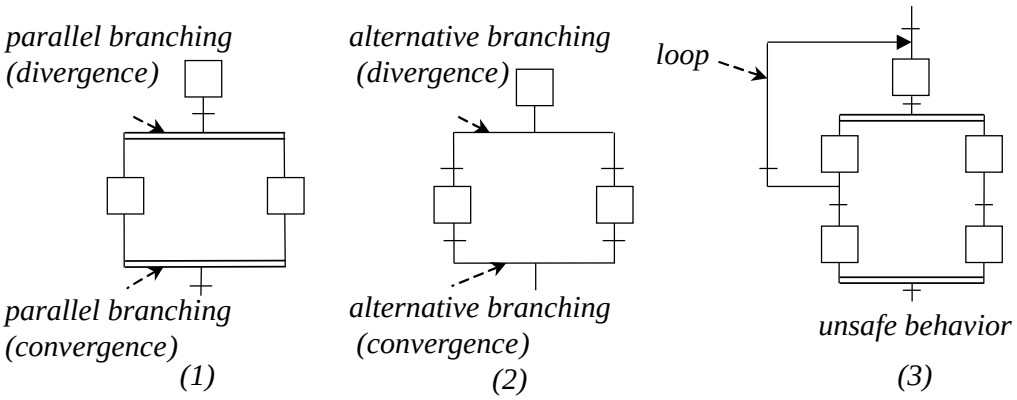

Figure 2. Different types of transitions

guards. Moreover, there is an order on actions which adapts to different execution models of parallelism. As shown in [4], the execution order of actions varies for different SFC programming tools, e. g., for one tool actions are executed according the alphabetic order of action names, for the other by user pre-defined orders and so on. This is covered by the partial order, which distinguishes the execution order on actions modifying the same variable. Note, if this order is empty all actions are non-ordered, i.e., executed in parallel.

An action block consists of an identifier for either a PLC program or a nested SFC and a qualifier q, where $q \in \{\mathtt{N}, \mathtt{S}, \mathtt{R}, \mathtt{P0}, \mathtt{P1}\}$. For a step s we denote by $block_a(s)$ the set of all action names of the action blocks associated to s and by $block_q(s)$ the set of all qualifiers. Moreover, in this setting we assume that all PLC programs are operating on Boolean variables. This is sufficient to illustrate how to model-check SFCs and simultaneously covers a lot of real-life SFC programs which reason about pumps, valves etc., which have just two states, e.g., either on or off.

The execution of SFCs is described by evolution rules similar to the firing rules of Petri nets. However, it is important to consider the SFC program execution on a PLC which differs from program execution on, e.g., an ordinary PC. In general PLC programs are executed in a cyclic manner. In every scan cycle first the new input from the environment (i.e., from the sensors of a plant such as pressure or temperature sensors) is read and stored. Then the PLC program is executed based on the stored input, i.e., the actions of the active steps are executed, which may change the output variables, and afterwards the transitions are taken. At the end of each cycle the output is sent to the environment, i.e., to the actuators of a plant such as valves and motors.

There are a couple of things worth mentioning when reasoning about SFCs and their semantics: First of all, in every cycle the actions are executed first and afterwards the guards are evaluated and the transitions are taken. In general, the actions are executed in a fixed order (e.g., lexicographical order on action names) given either explicitly or implicitly. Moreover, whenever a nested transition as well as a more top level transition are enabled, the top level transition is taken as well as the nested transition. However, the nested SFC will become inactive and its current location is stored as a *history step* from where control will resume the next time this SFC is activated. All steps that are "active" in a cycle, which means their actions will potentially be executed, are called *active steps.* The union of history steps and active steps are called *ready steps.* The actions which are potentially executed in a cycle are called *active actions* and the ones which have been activated by an S-qualifier and not yet reset are called *stored actions.*

A formal model and operational semantics can be found in [2]. The operational semantics relies on *configurations* describing a system state. A configuration is defined as follows:

Definition 2 (Configuration) *A* configuration *of an SFC and its sub-SFCs is a 5-tuple* $(\sigma, readyS, activeS, activeA, storedA)$*, where*

- σ *is the state of the variables,*
- $readyS$ *is the set of ready steps,*
- $activeS$ *is the set of active steps,*
- $activeA$ *is the set of active actions, and*
- $storedA$ *is the set of stored actions, i. e., the ones which might remain active outside the step they were called.*

Such a configuration is modified in the cycles of a PLC. In a cycle the following sequence is performed:

1. Get new input from the environment and store the information into the state of the variables σ.
2. Execute the set $activeA$ of active actions and update σ accordingly.

3. Determine the set of next $readyS$, $activeS$, $activeA$, and $storedA$.
4. Send the outputs to the environment by extracting the required information from the new state σ.

For each cycle the new active steps are the old ones plus the targets of the taken transitions, but without their source steps. Moreover, the new active steps, active actions and stored actions are computed recursively on the structure of the SFC. The semantics of an SFC is given by its possible set of configuration sequences. A configuration sequence consists of a finite or infinite number of configuration transformations, where every PLC cycle corresponds to one configuration transformation.

3. Translation to CaSMV

CaSMV is a symbolic model-checker which supports the verification of temporal logic properties of Kripke structures. The transition relation of a Kripke structure is expressed in CaSMV by evaluation rules depending on the current and the next state of each system variable q, i.e., `q` and `next(q)` in CaSMV notation. In order to translate an SFC to CaSMV we mimic the transition relation on a configuration of the SFC semantics. Remember, after reading the inputs, at every PLC cycle we first execute the actions and then evaluate the transitions and determine the new active steps and actions. Hence, whenever reasoning about guards or the new state of steps and actions we refer to the `next` state of a variable.

In the beginning we do not consider any order on actions and transitions. We assume that the activity of an action directly corresponds to an output variable, which is often the case as many actions only consist of opening and closing valves, switching motors on and off etc. In this case we do not need an order on actions, because the actions do not share variables. Furthermore, we first have no order of transitions which allows us to additionally check for conflicting transitions automatically. In Section 3.3 we show how to extend this framework by embedding orders on transitions and actions, which results into a deterministic execution model. This enables us to deal with more complex actions and situations were one variable is modified by more than one action and the execution order comes into play.

3.1. Data Structure of the CaSMV Module

A system modelled in CaSMV can be composed from components called *modules*. We use one module to describe the SFC and its actions and allow further modules to describe the environment or parts of the environment.

In order to translate an SFC $\mathcal{S} = (S, A, s_0, T, block, \sqsubset, \prec)$ to CaSMV we introduce the following Boolean variables:

- `ready_s`*i* for each step s_i, i.e., one variable for each step of the top level SFC and its hierarchically nested ones. These variables model whether the respective step is ready, this means, the step is active or control resides in it and waits to resume.
- `guard`*i* for each guard g_i. This variable represents the transition condition and is in general a Boolean expression reasoning about program variables and *input variables* `input`*i* (e.g., process variables from the plant to be controlled) and the *activity of steps* `step.X`*i*. Where, e.g., `step.X`*1* evaluates to true whenever step one is active.
- `active_a`*i* for each action a_i. This variable is introduced to code whether an action is active or not. This action might be an SFC itself.
- `stored_a`*i* for each action a_i, which indicates whether an action is currently stored, i.e., has been activated in the current or a previous step by an `S` qualifier.

Note, if we want to reason about the activity of a step s_i belonging to a nested SFC a_k, the variable `step.X`*i* will statically be substituted by $\texttt{ready\_s}i \wedge \texttt{next}(\texttt{active\_a}k)$. This means, a step s_i is active, if it is currently ready and after the execution of all actions the SFC it is nested in becomes (or remains) active.

Furthermore the CaSMV module has *input parameters* for each Boolean input variable of the SFC program. The behavior of the input variables is a priori chaotic, i.e., they might take any possible value, unless not otherwise specified. This allows to check an open system. Any restrictions on the behavior of input variables can be modelled in an additional CaSMV module representing the environment.

3.2. Evolution of State Variables

In this section we define how to code the transition relation on the variables defined in the previous section. This is of special interest for the activity of actions, which are tagged by qualifiers. Therefore, we explicitly define the `next`-state of all variables, but guards and input variables, since inputs are provided by some environment and the truth values of guards are determined by the evaluation of the Boolean expressions they represent.

Ready steps. The ready variable `ready_s`*i* of a step s_i is true if and only if there is a transition taken into s_i or it is already true and there is no transition taken out off s_i. In

case of a nested SFC it is additionally required that the SFC-action (the nested SFC itself) is active. In detail, for a nested SFC given by an action a_k the variables ready_s*i* for each *step* $s_i \in a_k$, are set to true only the SFC itself is active, i.e. active_a*k* holds, one of the preceding guards will hold after the executions of the actions, i.e., in the next cycle, and the corresponding source steps are currently ready.

Active actions. The value of active_a*k* for the activity of an action a_k depends on the activity of the steps s_j, where $a_k \in block_a(s_j)$, and the qualifiers tagged to a_k. The expression for determining next(active_a*k*) is defined by (act_N_S_steps $\vee$ act_P1_steps $\vee$ act_P0_steps $\vee$ stored_a*k*) $\wedge \neg$act_R_steps

where

- act_N_S_steps is $\bigvee_{\{s_j \mid a_k \in block_a(s_j)\}}$(next(ready_s*j*) $\wedge$ next(active_a*l*)) where a_k is tagged with the N or S qualifier and a_l is the SFC to which s_j belongs,
- act_P1_steps is $\bigvee_{\{s_j \mid a_k \in block_a(s_j)\}}$($\neg$ready_s*j* $\wedge$ next(ready_s*j*)) where a_k is tagged with the P1 qualifier,
- act_P0_steps is $\bigvee_{\{s_j \mid a_k \in block_a(s_j)\}}$(ready_s*j* $\wedge$ next($\neg$ready_s*j*)) where a_k is tagged with the P0 qualifier, and
- act_R_steps is $\bigvee_{\{s_j \mid a_k \in block_a(s_j)\}}$(next(ready_s*j*) $\wedge$ next(active_a*l*)) where a_k is tagged with the R qualifier and a_l is the SFC s_j belongs to.

This means, an action will become active if one of the following conditions hold: The step the action is associated to will become active and the action itself is tagged with either the N or the S qualifier, if a step the action belongs to will be entered in the next cycle and the action is tagged with the qualifier P1, if the step the action belongs to is active and will be inactive in the next cycle and the action is tagged with the qualifier P0, or the action is a stored one (see below). However, resetting an action has always priority and, thus, will in any case disable the activation.

Stored actions. The value stored_a*k* is set to true, if one or more steps where a_k is associated to are active and a_k is tagged with an S qualifier and there is no matching reset. It is set to false, whenever a matching reset action is called. Thus the next value of stored_a*k* is defined by next(stored_a*k*) = (act_S_steps $\vee$ stored_a*k*) $\wedge$ $\neg$act_R_steps where

- (act_S_steps is $\bigvee_{\{s_j \mid a_k \in block_a(s_j)\}}$(next(ready_s*j*) $\wedge$ next(active_a*l*)) where a_k is tagged with the S qualifier and a_l is the SFC s_j belongs to and
- act_R_steps as defined above.

Initialization. The *initial ready step* s_0 of the top-level SFC is initialized to true, denoting that this step is active at the beginning. All other steps are initially set to false. For reasons of simplicity we assume, that the initial step of the top level SFC contains no nested SFCs. This does not limit the set of translatable SFCs, because each SFC can be transformed into one meeting this constraint. Furthermore, all variables coding if the action is active or the action is a stored one are initially false.

3.3. Extension to Orders on Actions and Transitions

The translation presented above does not take into account the orders on actions, $\sqsubset$, and on transitions, $\prec$. Furthermore it only works for actions whose activity is mapped to an output variable. However, this approach can be extended to consider existing orders on transitions and actions. To take the order on transitions into account we modify the guards of the transitions such that there are no more conflicts. This can be done statically by adding constraints such that a transition is enabled iff its guard holds and no other higher-priority transition which shares at least one common source steps is enabled.

To consider more complex actions which make it necessary to deal with the order on actions we introduce for each output variable which is modified by more than one action a CaSMV state variable output*i* and a kind of microcycle. To realize the microcycle we define a CaSMV state variable which has as many states as there are actions modifying the output variable. Each state then corresponds to an action, and the variable output*i* evolves from one state to the next according to the given order. The output variable output*i* then is changed according to the state of the microcycle and the expression modelling the action's modification of the output variable. Furthermore we then also need a kind of global cycle for synchronization of all microcycles and the execution of the remaining actions.

4. Application to a Chemical Plant

In this section we apply the presented approach to a laboratory plant which was designed and built at the Process Control Laboratory of the University of Dortmund to serve as a test-bed and a demonstration medium for control and scheduling methods for multi product batch plants [3]. Compared to batch plants of industrial scale this plant is still of moderate size although it is already complicated enough to pose complex scheduling and control tasks.

In the plant two products are generated from three raw materials in three reactors simultaneously. The plant itself may be seen as part of a production line, therefore, there are buffer tanks for the three raw materials which are assumed

to come from a preceding step of this production line. Additionally there are buffer tanks for the products which are supposed to be consumed in a following production step.

For illustration purposes we only focus on one part of the plant, the production of one product in one of the reactors.

4.1. Example Process and Control Program

In Fig. 3 the reactor T3 is depicted, which is used to produce the chosen product, referred to as C, from two raw materials, referred to as A and B. The tanks T1 and T2 are used as buffer tanks for A and B respectively. They can be filled via the valves V1 and V2. The reaction is carried out by first filling A into T3 via valve V3. After this the contents of tank T2 is given into the reactor via V4, now B immediately reacts with A to C. Product C then directly can be withdrawn via V5 for further processing. Tanks T1 and T2 are equipped with sensors LIS+ for detecting the upper liquid level threshold and LIS− for the detecting that the tank is empty. Apart from an LIS− sensor tank T3 is also equipped with a stirrer M, which is used to ensure a homogenous solution within tank T3 during the reaction.

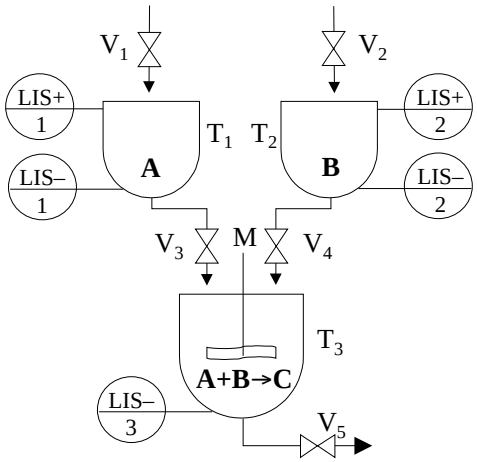

Figure 3. The plant

In Fig. 4 a control program for carrying out the reaction is shown. The control program consists of a kind of master-SFC, which allows the following three processes to run in parallel

- *filling T1 with A* given by action a_1 in step s_2,
- *filling T2 with B* given by action a_2 in step s_5 and
- *reaction in T3 and emptying T3* given in step s_7 as action a_3.

Action a_3 is given by an individual SFC, since this process itself has a certain complexity. As we have conflicting processes, such as *emptying contents of T1 into T3*, which is a sub step of a_3 and *filling T1 with A* we have the waiting steps s_1, s_4 and s_{10} which shall ensure that certain conditions (given as guards) hold before the processes start.

Apart from a_3 the actions are very simple. For these the value of actions activity simply determines the value of an output variable, e.g. the valve V1 is as long open, as a_1 is active.

Note, the control routines presented differ from those given in [3] as they are adapted to the excerpt of the plant presented.

4.2. Translation

The translation of the control program into CaSMV code follows directly from the definitions of Sect. 3. Here, we give just two examples of how to define the transition relation on state variables. The CaSMV code for these examples are at Fig. 5, where the symbols '`&`', '`|`', '`~`' (as well as '`!`') denote logical 'and', 'or' and 'not'. For the whole example the code can be found at one of the author's web pages[1].

Ready Steps. Step s_{12} of the nested SFC will become ready, if the preceding step s_{11} is currently ready, the SFC it is nested in is active and the guard *LISminus1* of the transition connecting these two steps will evaluate to true. On the other hand step s_{12} will become inactive, if it is currently ready, its SFC is active and the outgoing transition condition will hold. If none of these cases is satisfied s_{12} will keep its current status.

Active Actions. Action a_5 will become active if either s_{11} will be active, i.e., s_{11} will be ready and a_3 will be active, or a_5 is stored and s_{13} is not an active step, since a_5 is reset in s_{13}. In any other case a_5 will be inactive.

Stored Actions. Action a_5 will be stored, if step s_{11} is active, i.e., step s_{11} is ready and action a_3 is active, and a_5 will be marked as not stored if it is reset, i.e., step s_{13} is active. In all other cases a_5 is unchanged.

4.3. Specification of Verification Tasks

For the given SFCs we check the following properties:

Reachability of each step. We check this to ensure that there is no unused code. The corresponding CTL specification added to the CaSMV input file is for a step s_i: `SPEC EF si`, i.e., there exists an execution path which will eventually reach s_i.

[1] `http://www.informatik.uni-kiel.de/~rhu/APAQS01/`

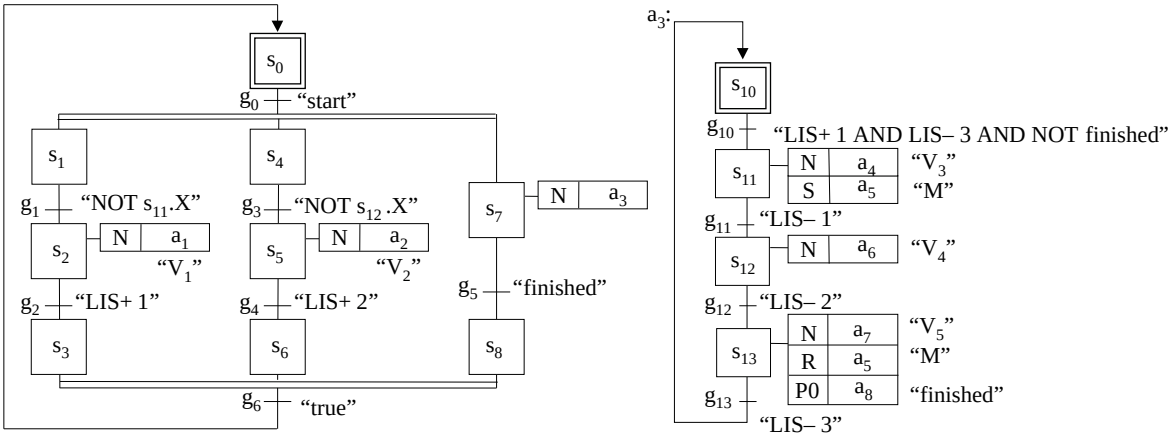

Figure 4. Control SFCs

Absence of deadlocks. We check that whenever a step s_i is reached it is possible to extend this run such that s_i is reached once more, i.e., infinitely often. In CaSMV this is specified by: `SPEC AG (AF si).`

Plant-specific requirements. For the design of control programs of batch plants the allocation of plant equipment to different production steps or processes is a central issue. There are often processes, which are in conflict because they need the same resources. E.g., *emptying contents of T1 into T3* and *filling T1 with A* are in conflict, because they both compete for tank T1. Therefore, it has to be checked that equipment is exclusively used by one process at a time. In the given example it has to be ensured that both valves for filling and emptying a tank are not open simultaneously. E.g., for tank T1 the valves V1 and V3 shall never be open at the same time, which leads to the specification: `SPEC AG !(V1 & V3).`

The verification tasks presented here are independent of a specific environment but reason about the control software only. In order to verify, e.g., that there is no overflow in a tank, parts of the plant and the environment have to be included into the model and checked in combination with the current controller.

4.4. Verification Results

All verification tasks presented above are checked within a fraction of one second on a SUN ULTRA 1. This is not surprising, since the model is still of small size and for illustration purpose only.

It was proven that every step is reachable and there are no deadlocks. We also verified that the tanks T1 and T3 are never filled and emptied at the same time. However, tank T2 does not fulfill this requirement. The counter trace produced by CaSMV shows that both valves V2 and V4 may be opened simultaneously. This happens, because it is only required when entering step s_5 that step s_{12} is not active (`NOT` s_{12}`.X`), i.e. that filling T2 does not start, if it is already in the emptying phase. However, when entering s_{12} there is no condition that checks, if the tank is still in the filling phase.

Hence, the verification detected a flaw in the control program which is not obvious to see and the counter trace helps to see why it happened and how to prevent it.

5. Conclusions

In this work we presented an automatic verification approach for SFCs by developing a translation from SFCs into the native language of CaSMV. Moreover, standard verification issues like the reachability of steps or the absence of deadlocks can be checked fully automatically, since when generating CaSMV code these safety requirements can be easily generated, too.

Furthermore, this work is the first one which comprises all the major features of SFCs and it is the only one taking the rather interesting concept of activity manipulation, i.e., action qualifiers, into account. Hence, it allows a real application to existing control software.

Future work will concentrate on the implementation of the presented method and its application to industrial size systems. However, additional measures will be necessary in order to cope with the complexity of the systems. One way to tackle this are decomposition techniques and the application of assumption/commitment methods [17, 13, 18, 1]. Moreover, an extension to timed SFCs and timed qualifiers is possible as well as extending the framework to data types with infinite domains like integers. This might on the one

```
default next(readyS_s12) := readyS_s12; in case{
   (readyS_s11 & next(LISminus1) & activeA_a3) : next(readyS_s12) := 1;
   (readyS_s12 & next(LISminus2) & activeA_a3) : next(readyS_s12) := 0;
}


default next(activeA_a5) := 0;
in case{
   next(readyS_s13) & next(activeA_a3) : next(activeA_a5) := 0;
  (next(readyS_s11) & next(activeA_a3)) | next(storedA_a5) : next(activeA_a5) := 1;
}


default next(storedA_a5) := storedA_a5;
in case{
   next(readyS_s13) & next(activeA_a3) : next(storedA_a5) := 0;
   next(readyS_s11) & next(activeA_a3) : next(storedA_a5) := 1;
}
```

Figure 5. CaSMV code for examples

hand lead to the application of real-time model-checkers like UPPAAL [5] or KRONOS [20] and on the other hand to the development of transformations into finite subclasses or the combination of model-checking with deductive methods.

Acknowledgements

This work has been partially supported by the European Community in the Esprit Long-Term Research Project 26270 – VHS (Verification of Hybrid Systems) and the German Research Council (DFG) in the special program *Integration of Software Specification Techniques in Engineering Applications* under grant LA 1012/6-1. Furthermore, we like to thank Ben Lukoschus for fruitful discussions.

References

[1] M. Abadi and L.Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, 1995.

[2] N. Bauer, R. Huuck, and B. Lukoschus. A parameterized semantics for sequential function charts. Institute of Computer Science and Applied Mathematics, University of Kiel, 2001.

[3] N. Bauer, S. Kowalewski, G. Sand, and T. Löhl. A case study: Multi product batch plant for the demonstration of control and scheduling problems. In S. Engell, S. Kowalewski, and J. Zaytoon, editors, *ADPM2000 Conference Proceedings*, pages 383–388, 2000.

[4] N. Bauer and H. Treseler. Vergleich der Semantik der Ablaufsprache nach IEC 61131-3 in unterschiedlichen Programmierwerkzeugen. In *GMA-Kongress 2001*, volume 1608 of *VDI-Berichte*, pages 135 –142. VDI-Verlag, 2001.

[5] J. Bengtsson, W.O.D. Griffioen, K. J. Kristoffersen, F. Larsson K. G. Larsen, P. Pettersson, and W. Yi. Verification of an audio protocol with bus collision using uppaal. In *Computer Aided Verification*, pages 244–256, 1996.

[6] F. Bonfatti, P.D. Monari, and U. Sampieri. *IEC 1131-3 Programming Methodology*. CJ International, Fontaine, France, first edition, 1999.

[7] S. Bornot, R. Huuck, Y. Lakhnech, and B. Lukoschus. Verification of sequential function charts using SMV. In *PDPTA 2000: International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, USA, June 26–29, 2000*, pages 2987–2993, 2000.

[8] E.M. Clarke and E.A. Emerson. Synthesis of synchronisation skeletons for branching time temporal logic. In D. Kozen, editor, *Workshop on Logic of Programs*, volume 131 of *LNCS*. Springer-Verlag, 1981.

[9] R. David and H. Alla. *Petri Nets & Grafcet*. Prentice Hall, 1992.

[10] K. Fujino, K. Imafuku, Y. Yamashita, and H. Nishitani. Design and verification of the SFC program for sequential control. *Computers and Chemical Engineering*, 24:303 – 308, 2000.

[11] International Electrotechnical Commission, Technical Committee No. 65. *Programmable Controllers – Programming Languages, IEC 61131-3*, second edition, November 1998. Committee draft.

[12] International Electrotechnical Commission, Technical Committee No. 848. *IEC 60848, Preparation of function charts for control systems*, 1992.

[13] C.B. Jones. *Development methods for computer programs including a notion of interference*. PhD thesis, Oxford University Computing Laboratory, 1981.

[14] S. Lampérière and J.-J. Lesage. Formal verification of the sequential part of plc programs. In R. Boel and G. Stremersch, editors, *Discrete Event Systems*, pages 247–254. Kluwer Academic Publishers, 2000.

[15] P. L'Her, J.L. Scharbarg, P. Le Parc, and L. Marce. Proving sequential function chart programs using automata. *LNCS*, 1660:149 – 163, 1998.

[16] K.L. McMillan. *The SMV Language*. Cadence Berkeley Labs, 1999. http://www-cad.eecs.berkeley.edu/kenmcmil/language.ps.

[17] J. Misra and K.M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(7):481–426, 1981.

[18] A. Pnueli. In transition for global to modular temporal reasoning about programs. In *Logics and Models of Concurrent Systems*, volume 13 of *NATO ASI-F*. Springer, 1984.

[19] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *5th International Symposium on Programming*, volume 137 of *LNCS*, pages 337–351. Springer-Verlag, 1982.

[20] Sergio Yovine. KRONOS: A verification tool for real-time systems. *International Journal on Software Tools for Technology Transfer*, 1(1-2):123–133, 1997.