Verification of PLC Programs given as Sequential Function Charts

Nanette Bauer[1], Sebastian Engell[2], Ralf Huuck[3], Sven Lohmann[2], Ben Lukoschus[4], Manuel Remelhe[2], Olaf Stursberg[2]

[1] BASF AG, 67056 Ludwigshafen, Germany
nanette.bauer@basf-ag.de
[2] Process Control Laboratory (BCI-AST)
University of Dortmund, 44221 Dortmund, Germany
{s.engell|s.lohmann|m.remelhe|o.stursberg}@bci.uni-dortmund.de
[3] National ICT Australia Ltd (NICTA),
The University of New South Wales, Sydney, Australia
rhuuck@cse.unsw.edu.au
[4] Institute of Computer Science and Applied Mathematics
University of Kiel, 24098 Kiel, Germany
bls@informatik.uni-kiel.de

Abstract. Programmable Logic Controllers (PLC) are widespread in the manufacturing and processing industries to realize sequential procedures and to avoid safety-critical states. For the specification and the implementation of PLC programs, the graphical and hierarchical language Sequential Function Charts (SFC) is increasingly used in industry. To investigate the correctness of SFC programs with respect to a given set of requirements, this contribution advocates the use of formal verification. We present two different approaches to convert SFC programs algorithmically into automata models that are amenable to model checking. While the first approach translates untimed SFC into the input language of the tool Cadence SMV, the second converts timed SFC into timed automata which can be analyzed by the tool Uppaal. For different processing system examples, we illustrate the complete verification procedure consisting of controller specification, model transformation, integration of dynamic plant models, and identifying errors in the control program by model checking.

Keywords. Analysis, Automata, Model Checking, Logic Control.

1 Introduction

A large part of the control software of processing and manufacturing systems performs logic and supervisory control. Logic control is characterized by the reaction of the controller to events generated by the plant (e.g., a relevant quantity exceeds a threshold), and the controller selects one out of finitely many control actions. The two major objectives of such controllers are (a) the realization of sequential procedures, as for example to establish a given sequence of production steps, and (b) to ensure a safe operation of the plant. The latter may involve to initiate an emergency routine if a malfunction or a deviation from the desired operation is detected.

While many industrial logic controllers are still implemented in the languages *instruction list*, *ladder diagram*, or *continuous function charts* [1], the so-called *Sequential Function Charts* (SFC) become increasingly important and accepted. By using SFC, which are standardized according to [2], the control logic can be specified in an intuitive way. Sequential, parallel, and nested procedures are represented graphically, and subfunctions given in any of the other languages listed above can be embedded. Irrespectively of the language chosen to model the controller, the correctness with respect to the intended behavior of the controlled system is, of course, crucial. This is most apparent for safety specifications, i.e., the objective of the logic controller is to prevent that the plant runs into a state which is harmful for the personnel, the equipment, or the environment of the plant. While it is industrial practice to rely on extensive testing to check that the controller is correct, academia has intensively studied the technique of formal verification for this purpose. It performs a manual or algorithmic proof that a logic controller complies with a set of formal requirements and has been investigated in, e.g., [3–6]. From the various known verification techniques [7], we focus on *model checking* [8] which (partially) computes the reachable set of a state-transition model and evaluates if a formal requirement expressed in temporal logic holds for this set.

In order to apply model checking to controllers given as SFC, the latter first have to be translated into a state-transition model. The approach in [9] uses Petri-Nets as the target format while the methods in [10, 11] transform the SFC into automata and apply model checking afterwards. This contribution follows the latter approach and describes three important extensions:

(a) We explicitly account for the cyclic operation mode of the hardware on which logic controllers are usually executed, i.e. of Programmable Logic Controllers (PLC). Each cycle of this mode consists of a scanning step (in which the inputs from the plant are read), the step of executing possible transitions of the SFC, and finally writing the outputs to the plant.

(b) We present transformation schemes to convert SFC into the input language of two different tools for model checking. The first scheme is applicable to SFC without real-time quantifiers. Such charts are transformed into the input format of the tool Cadence-SMV [12] which is known to be efficient for large finite-state automata [13]. The second approach considers real-time specifications of the control program by transforming the SFC into timed automata using a procedure based on graph grammars. To verify timed automata, the tool Uppaal is applied [14].

(c) For processing and manufacturing systems, many requirements are usually formulated for the controlled plant, i.e., it is not sufficient to consider only a model of the controller for verification, but one has also to consider the plant behavior. For the two approaches listed above, we describe how an appropriate model of the plant behavior (specified either as a finite state automaton or a timed automaton) can be used to verify whether the controlled plant shows the intended behavior.

2 Verification Objectives and Modeling Alternatives

Figure 1 summarizes our overall design procedure for logic controllers: The controller is constructed as an SFC in a manual design procedure in which a specification of the control goals and the expected plant behavior are taken into consideration. This step involves to formulate a sequence of control actions that realize the goals given an intuitive understanding of how the plant reacts to these actions. Depending on whether the controller includes timed actions, the SFC is translated into a finite state automaton (FSA) or a timed automaton (TA). The analysis tool (optionally) composes the controller with a formal model of the plant and checks the validity of a formalized representation of the requirements. The plant model is also represented as FSA or TA, depending on whether quantitative time is relevant for the analysis task. If the analysis reveals that the requirements are met, the SFC can be transferred to the PLC. A violation of the requirements may either be due to a wrong controller design (i.e., the SFC has to be modified) or to an insufficiently detailed plant model (i.e., a less conservative one has to be employed).

In order to illustrate the choice of a plant model and a typical set of requirements, we consider the simple processing system shown in Fig. 2: The plant consists of two tanks *T1* and *T2* with heating devices *H1* and *H2*, a condenser *C1*, a pump *P1*, four on-off valves *V1* to *V4*, and sensors for monitoring if thresholds for the liquid levels (*LI*), the temperature (*TI*), the concentration (*QI*), and the flow (*FI*) are exceeded. The nominal operation of this system (and thus a control goal) is as follows: *T1* is first filled through *V1* with a liquid that contains a dissolved substance. The liquid is heated up in *T1* by the heater *H1* until the boiling point is reached. By further supplying heat, a certain amount of solvent is evaporated until the concentration of the liquid has reached a desired concentration. During the evaporation, vapor is condensed in *C1* which is cooled by a cooling agent that is supplied through *V4*. When the evaporation is finished, the liquid is transferred from *T1* into *T2* through *V2*. This procedure is repeated

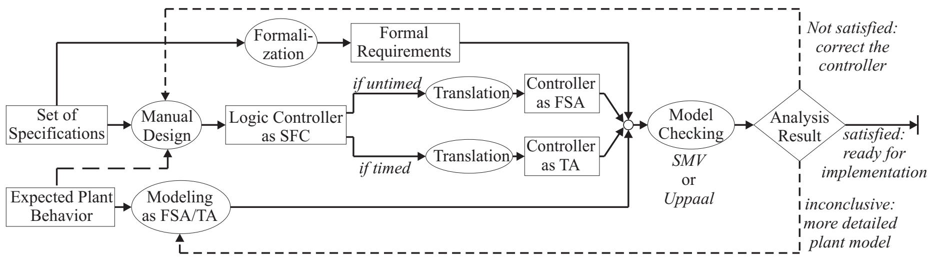


Fig. 1. The controller design procedure with: TA - timed automaton, FSA - finite state automaton, SFC - sequential function charts.

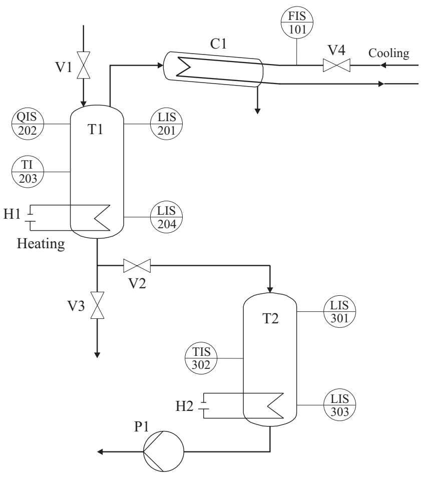


Fig. 2. Flowchart of an evaporator system.

twice until *T2* is filled with three batches from *T1*. The content of *T2* (the product) is then pumped out of *T2* through *P1*, and afterwards the complete operation can start again. In addition, two disturbance scenarios are considered, an appropriate handling of which constitutes two further control objectives: (a) In the event of a cooling failure (detected by *FIS101*) the evaporation is stopped after a short period of time (to avoid overpressure) and, if the concentration goal is not reached by then, the content of *T1* is disposed through *V3*. (b) In the event of a heating failure, *T1* is also emptied immediately through *V3*, since the process control goal cannot be achieved in any case. In both cases the nominal operation should be resumed when the faulty devices have been repaired or replaced.

A possible SFC controller as a result of manual design is shown in Fig. 3. Each step is denoted by a rectangle and a step identifier (*S0* is the initial step). The transition between two consecutive steps (marked by a bold horizontal line) carries a condition given as a Boolean expression. If the latter evaluates to *true*, the transition can be taken and the following step becomes active. The variables that occur in the conditions are either input variables (i.e., their values represent information received from the plant) or internal variables (e.g. *count*). The actions assigned to the steps are specified in action blocks by a qualifier and a Boolean variable. The variables that are manipulated by action blocks are either internal or output variables. The latter represent the control actions that are transmitted to the plant. The two branches enclosed by the horizontal double lines represent simultaneous operations, where the left branch accounts for the nominal operation and the right one for the failure scenarios. In Sect. 3, the syntax and semantics of SFCs is described in more detail, and Sect. 5 contains a description of how the SFC in Fig. 3 realizes the desired operation of the evaporator.

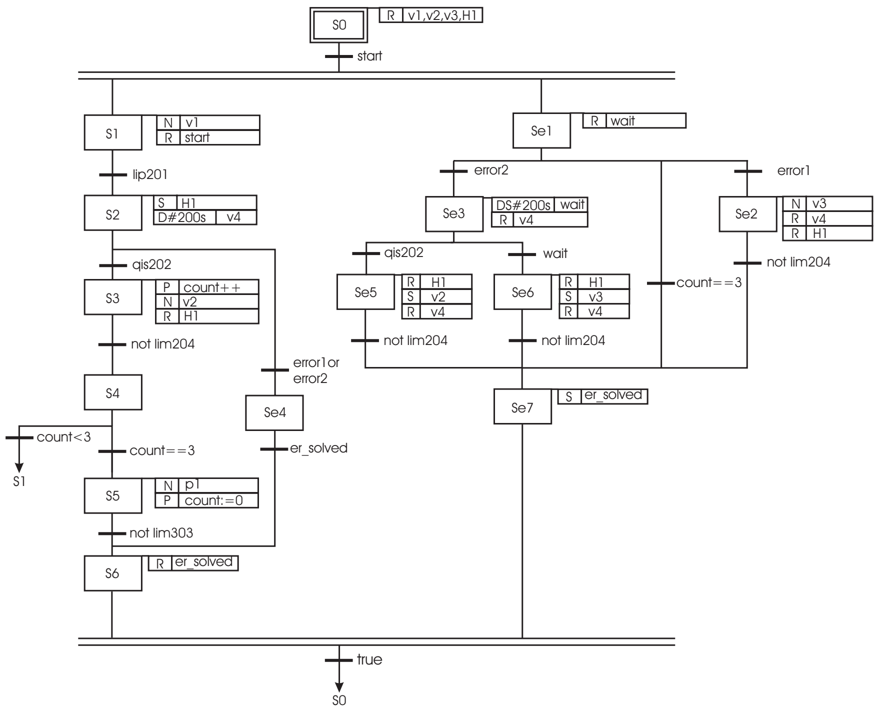


Fig. 3. SFC-controller for the evaporation system.

For systems like the controlled evaporator, the verification usually aims at checking requirements that are of the following type: (a) it has to be checked whether the controller indeed realizes the desired production sequence; (b) safety guidelines imply that unsafe states (as a maximum or minimum temperature in *T1*) are never reached, and (c) the SFC must never be deadlocked. The first two requirements can obviously only be checked if a plant model is employed that represents the behavior of quantities like levels, temperatures, and concentrations. The last requirement should be checked for arbitrary values of the input variables, i.e., a plant model is not required for (c). Section 5 describes the verification of the first two requirements for the evaporator system, while Sect. 3 deals with the structural analysis of SFC.

3 Analysis of Untimed SFC Programs

This section describes the algorithmic verification of SFC programs without time quantifiers using the model checker Cadence SMV (CaSMV). While the method proposed in Sect. 4 is, of course, also applicable to untimed SFC, we deem it preferable to use CaSMV in this case due to the known efficiency of symbolic model checking for untimed models.

3.1 The SFC Language and Semantics

Sequential function charts are described in the IEC 61131-3 standard [2] as elements of a graphical programming and structuring language for PLCs, and the syntax and semantic is formally defined in [15]. For an SFC $\mathcal{S}$, this syntax introduces the symbol S for its sets of steps with an initial step s_0 and a function *block* which assigns a set of action blocks to each $s_i \in S$. An action block is a pair (a, q), where a is an action name and q is one of the following action qualifiers. We only consider the untimed action qualifiers `N` (non-stored), `R` (reset), `S` (set or stored), `P1` (pulse, rising edge), and `P0` (pulse, falling edge) in this section. While non-stored actions are active only when its corresponding step is activated, stored actions continue being active until a reset action is executed. Actions with the pulse qualifier are performed only once when entering (`P1`) or exiting (`P0`) a step. If the action name is a Boolean variable, the variable is *true* if the action is active and *false* otherwise. Action qualifiers control the activity of the respective action depending on the activity of steps. We assume that the SFC operates only on Boolean variables. Action names refer to a Boolean variable, a subordinated SFC (thus enabling nested or hierarchical structures) or programs written in one of the other programming languages defined in the standard.

The execution of SFC is described by evolution rules similar to the firing rules of Petri nets considering the cyclic operation of PLC as mentioned in Sect. 1, i.e. the actions are executed first in each cycle, and the guards are evaluated and the enabled transitions are taken afterwards. In general, the actions are executed in a fixed order given either explicitly or implicitly. Whenever a nested SFC gets deactivated, its enabled transitions are still taken in that cycle, but then the nested SFC becomes inactive and its current location is marked as a *history step* from which the executions resumes if this SFC is activated again. All steps that are "active" in a cycle (meaning that their actions are executed) are called *active steps.* The union of history steps and active steps is 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 which have not yet been reset are called *stored actions.*

The formal operational semantics for SFC according to [15] is based on *configurations* describing a system state as follows:

Definition 1 (Configuration). *A* configuration *of an SFC and its sub-SFC is a 5-tuple $(\sigma, readyS, activeS, activeA, storedA)$, where σ is the state (i.e., a function assigning a value to each variable), and readyS denotes the set of ready steps, activeS the set of active steps, activeA the set of active actions, and storedA the set of stored actions.*

Such a configuration is modified within a PLC cycle as follows: (1) get new input from the environment and store the information in σ; (2) execute the set *activeA* of active actions and update σ accordingly; (3) determine *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 [15]. The semantics of an SFC is given by its possible set of configuration sequences. A configuration sequence consists of a possibly infinite number of transformations of configurations, where each PLC cycle corresponds to one transformation.

3.2 Translation to CaSMV

CaSMV [12] is a symbolic model-checker [16, 17, 8] which supports the algorithmic 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 (`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. We initially assume that each action changes an output variable—in this case, an explicit ordering of the actions is not necessary since actions do not share output variables. We also start without an explicit order of transitions which allows us to additionally check for conflicting transitions automatically. Later we show how to extend this framework by embedding orders on transitions and actions resulting in a deterministic execution model. This enables us to deal with more complex actions and situations where a variable is modified by more than one action.

Data Structure of the CaSMV Module. A system modeled in CaSMV can be composed from components called *modules*. One module describes the SFC and its actions, and further modules may describe the environment or parts thereof. The translation from a system of SFC into a CaSMV module requires 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 the subordinated ones. These variables model whether the respective step is ready, i.e. 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 formulated over 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 s_1 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 can be an SFC itself.
– `stored_a`*i* for each action a_i, which indicates if an action is currently stored, i.e., it has been activated in the current or a previous step by an `S` qualifier.

A 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 the SFC program as an open system. Any restrictions on the behavior of input variables can be modeled in an additional CaSMV module representing the environment.

Evolution of State Variables. Next we define how to code the transition relation on the variables defined above. 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, except for guards and input variables, since inputs are provided by the environment and the truth values of guards are determined by the evaluation of the Boolean expressions which they represent.

Ready steps. The ready variable $\texttt{ready\_s}i$ 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 leaving s_i. Inside a nested SFC, transitions can only be taken if the nested SFC itself is active. In detail, for a nested SFC given by an action a_k, the variable $\texttt{ready\_s}i$ for each step s_i of a_k can only be changed if $\texttt{active\_a}k$ holds.

Active actions. The value of $\texttt{active\_a}k$ for the activity of an action a_k depends on the activity of the steps s_j with $(a_k, q) \in \mathit{block}(s_j)$, and the qualifier q tagged to a_k. The expression for determining $\texttt{next(active\_a}k\texttt{)}$ is defined by $(\texttt{act\_N\_steps} \vee \texttt{act\_S\_steps} \vee \texttt{act\_P1\_steps} \vee \texttt{act\_P0\_steps} \vee \texttt{stored\_a}k) \wedge \neg\texttt{act\_R\_steps}$ where

- $\texttt{act\_N\_steps} = \bigvee_{\{\texttt{s}_\texttt{j} \mid (\texttt{a}_\texttt{k},\texttt{N}) \in \mathit{block}(\texttt{s}_\texttt{j})\}} (\texttt{next(ready\_s}j\texttt{)} \wedge \texttt{next(active\_a}l\texttt{)})$,
- $\texttt{act\_S\_steps} = \bigvee_{\{\texttt{s}_\texttt{j} \mid (\texttt{a}_\texttt{k},\texttt{S}) \in \mathit{block}(\texttt{s}_\texttt{j})\}} (\texttt{next(ready\_s}j\texttt{)} \wedge \texttt{next(active\_a}l\texttt{)})$,
- $\texttt{act\_P1\_steps} = \bigvee_{\{\texttt{s}_\texttt{j} \mid (\texttt{a}_\texttt{k},\texttt{P1}) \in \mathit{block}(\texttt{s}_\texttt{j})\}} (\neg\texttt{ready\_s}j \wedge \texttt{next(ready\_s}j\texttt{)})$,
- $\texttt{act\_P0\_steps} = \bigvee_{\{\texttt{s}_\texttt{j} \mid (\texttt{a}_\texttt{k},\texttt{P0}) \in \mathit{block}(\texttt{s}_\texttt{j})\}} (\texttt{ready\_s}j \wedge \texttt{next(}\neg\texttt{ready\_s}j\texttt{)})$, and
- $\texttt{act\_R\_steps} = \bigvee_{\{\texttt{s}_\texttt{j} \mid (\texttt{a}_\texttt{k},\texttt{R}) \in \mathit{block}(\texttt{s}_\texttt{j})\}} (\texttt{next(ready\_s}j\texttt{)} \wedge \texttt{next(active\_a}l\texttt{)})$.

(In the definitions above, a_l denotes the SFC to which s_j belongs.)

Thus, an action will become active if one of the following conditions hold: the step with which the action is associated becomes active and the action itself is tagged with the qualifier `N` or `S`; a step the action belongs to will be entered in the next cycle and the action is tagged with the qualifier `P1`; 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). Resetting an action always has higher priority and, thus, will in any case deactivate a_k.

Stored actions. The value $\texttt{stored\_a}k$ 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 $\texttt{stored\_a}k$ is defined by $\texttt{next(stored\_a}k\texttt{)} = (\texttt{act\_S\_steps} \vee \texttt{stored\_a}k) \wedge \neg\texttt{act\_R\_steps}$.

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 SFC. This does not limit the set of SFC that can be translated, because each SFC can be transformed into one that meets this constraint. Furthermore, all variables encoding that an action is active or stored are initially *false*.

Extension to Orders on Actions and Transitions. The translation presented above does not consider orders on actions and on transitions. Furthermore, it only works for actions which map their activity to an output variable. However, this approach can be extended to consider 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 if and only if its guard holds and no other higher-priority transition which shares at least one common source step is enabled.

To consider more complex actions which make it necessary to deal with the order on actions we introduce a new CaSMV variable `output`i for each output variable which is modified by more than one action. Each of these new variables is modified in a *micro-cycle* by all actions which access this output variable, while using the correct action ordering. Furthermore, we need a *global cycle* for the synchronization of all micro-cycles and for the execution of the remaining actions as described above.

3.3 Example: Application to a Chemical Plant

The presented approach is applied to a batch laboratory plant in which two products are simultaneously produced from three raw materials in three reactors [18]. We focus here only on the production of one product in one of the reactors.

Process and Control Program. Figure 4 shows the reactor T_3 used to form the product C from two raw materials, referred to as A and B. The tanks T_1 and T_2 are used as buffers for A and B, and they are filled through the valves V_1 and V_2. The production procedures starts by filling A into T_3 through V_3, and afterwards the contents of T_2 are filled into T_3 through V_4. B immediately reacts with A to C, and the product C can be withdrawn through V_5 for further processing. The vessels are equipped with sensors LIS+ and LIS− for detecting that upper and lower threshold for the liquid levels are crossed. T_3 is additionally equipped with a stirrer M.

Figure 5 contains a control program consisting of a top-level SFC which triggers the following three parallel processes: (a) *filling* T_1 *with A* given by action a_1 in step s_2, (b) *filling* T_2 *with B* given by action a_2 in step s_5, and (c) *reaction in* T_3 *and emptying* T_3 given in step s_7 as action a_3. The action a_3 is given as a separate SFC. Due to conflicting processes, such as "empty contents

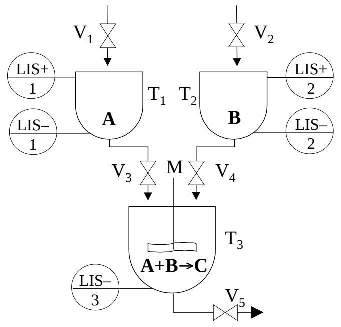


Fig. 4. A part of the multi-product batch plant

of T_1 into T_3" (a sub-step of a_3) and "fill T_1 with A", the waiting steps s_1, s_4 and s_{10} are included to ensure that certain conditions (given as guards) hold before the processes start. Apart from a_3, the actions are very simple since the activity simply determines the value of an output variable, e.g., V_1 is open as long as a_1 is active.

Translation to CaSMV. The translation of the control program into CaSMV code follows directly from the definitions in Sect. 3.2. Figure 6 shows two examples for defining the transition relation on state variables, where the CaSMV code contains the symbols '`&`', '`|`', (and '`!`') denoting the logical 'and', 'or' (and 'not'). The 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 "LIS− 1" 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. In any other case, s_{12} keeps its current value. The action a_5 will become active if either s_{11} is active (i.e., s_{11} is

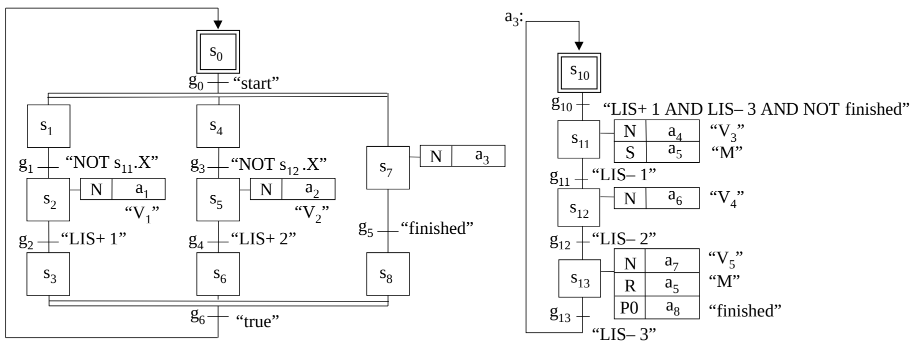


Fig. 5. Control SFC for the production in reactor T_3

```
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;}
```

Fig. 6. CaSMV code fragments for the SFC a_3

ready and a_3 active) or a_5 is stored and s_{13} is not active. In any other case, a_5 will be inactive. Furthermore, a_5 is stored if s_{11} is active, and a_5 is not stored if it is reset in s_{13}.

Specification of Verification Tasks. The translated SFC is checked for the following properties: (a) reachability of each step to ensure that the SFC does not contain unused code; in CaSMV the corresponding CTL specification is for a step s_i: `SPEC EF s`*`i`*, i.e., there exists an execution path by which s_i is eventually reached; (b) the absence of deadlocks by checking that each run by which s_i is reached can be extended such that s_i is reached once more: `SPEC AG (AF s`*`i`*`)`; and (c) plant specific requirements: For batch plants, the conflicting allocation of equipment by different production steps is often important; e.g. the steps "emptying contents of T_1 into T_3" and "filling T_1 with A" are in conflict since they compete for tank T_1. Therefore, it has to be verified that each piece of equipment is exclusively used by one process at a time. As an example, we check for tank T_1 that the valves V_1 and V_3 are never open at the same time, specified by: `SPEC AG !(V1 & V3)`.

The verification tasks presented here are independent of a specific environment, they 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 have to be checked in combination with the controller.

Verification Results. All verification tasks presented above are checked within a fraction of a second on a Sun UltraSPARC 1. This is not surprising, since the model is still of small size and for illustration purpose only. It is verified that every step is reachable and there are no deadlocks. We also verified that the tanks T_1 and T_3 are never filled and emptied at the same time. However, tank

T_2 does not fulfill this requirement. The counter trace produced by CaSMV shows that both valves V_2 and V_4 may be open simultaneously. This happens because when entering step s_5 it is only required that step s_{12} is not active (`NOT` s_{12}`.X`), i.e., that filling T_2 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.

4 Model checking of timed SFC

In timed SFC, time specifications in the transition conditions and actions have to be considered. Timed action qualifiers can be recognized by the letter `D` for delayed actions and `L` for time-limited actions. Both can be combined with the "set" qualifier `S`. [2] defines five timed qualifiers: `L`, `D`, `LS`, `SD`, and `DS`. Timed transition conditions contain inequality expressions that compare a timer variable with a real-valued expression. In most cases, timer variables reference step timers which store the time elapsed since the corresponding step was activated the last time. Step timers are denoted by the step name extended with the suffix ".T". Finally, the PLC cycle itself affects the timed behavior of an SFC.

4.1 Timed Automata and Uppaal

In order to check the timing properties of a given SFC, it has to be transformed into a formalism that enables appropriate timed analysis based on automatic verification software. The timed automaton (TA) formalism satisfies this requirement and is used here. The graphical representation of TA consists of nodes that are called locations, and directed arcs that represent the discrete transitions [19]. The current state of a timed automaton is given by the current location together with the valuation of integer and clock variables. The valuations of all clock variables grow with the same rate corresponding to the progress of time; the only way to influence a clock variable is to reset it to zero by a transition assignment. Informally, the semantics of a TA can be understood such that (i) the TA can stay at most as long in the current location as an *invariant* (a condition for the clock values) is satisfied, (ii) a transition can be taken when a condition for the clock values called *guard* is fulfilled, and (iii) a transition can reset clocks.

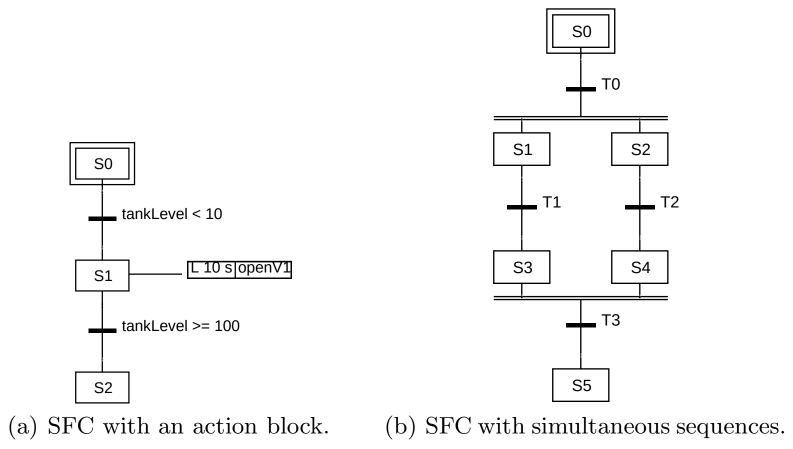
We refer to the specific form of TA used within the verification tool Uppaal [20, 21]. In the Uppaal language, a model consists of a collection of timed automata that can communicate via shared variables and channels. Channels are used to synchronize the processes, i.e., certain transitions of different automata can be forced to be taken synchronously. The channels have to be declared globally and are referenced in the synchronization labels of those transitions that are synchronized. In the synchronization label the name of the channel has to be followed either by an exclamation mark or by a question mark, indicating a sending or receiving role of the transition. Only two transitions can synchronize at a time using a binary channel. For this, one transition has to be sender and the other has to be receiver on the same channel. Non-deterministic situations occur when several senders and receivers may use a channel at the same time. Other elements specific to Uppaal such as broadcast channels, urgent locations and committed locations will be explained in the context of the representation of SFC. For a formal definition of the Uppaal language we refer to [19, 22].

4.2 Representation of SFC in Uppaal

The Uppaal tool includes a graphical user interface for modelling TA and for an interactive animation of the behavior. To make use of these features in verifying SFC programs, it is necessary to convey the structure of the SFC as far as possible to the TA domain, thus to enable the user to identify certain SFC components in the TA model. In the case of an SFC without parallel branches, as shown in Fig. 7(a), the complete structure of steps and transitions can be reproduced by the locations and the transitions of a single automaton. This even applies to complex SFC including nested loops and alternative branches. However, parallel branches as shown in Fig. 7(b) cannot be represented in one automaton such that the structure is preserved. Therefore, a connected group of parallel sequences is represented by one location in the embedding automaton, and additional automata represent the parallel branches. This will be explained in detail below. Note that the locations mentioned above do not represent the activity of steps but determine which steps are ready, i.e. they represent the union of history steps and active steps. To mark the set of active steps, additional Boolean variables are used with names which are composed of the step name and the suffix "_X".

The set of active actions is also represented by Boolean variables with names formed of the action name and the suffix "_Q". In the standard, a logic diagram including flip-flops, timers, and logical operations, defines how the value of an action variable has to be computed. The circuit can be divided into sections that independently model the dynamic behavior of the qualifiers `P0`, `P1`, `S`, `L`, `D`, `LS`, `SD`, and `DS`, and a section that describes the superposition of the results together with the qualifiers `N` and `R`. For each qualifier, a Boolean input denotes whether the qualifier is in use by the currently active steps or not. We use integer variables for modeling this. The name of such a variable is the concatenation of the action name and the qualifier symbol. The value of the variable determines the number of currently active steps that use the given combination of action and qualifier. The qualifier sections and the superposition section of the logic diagram are modelled by individual timed automata. Hence, for each action at least one and at most nine automata have to be instantiated depending one the qualifiers used in the SFC program.

Finally, we have to consider the PLC cycle semantics. This is achieved by an automaton that forces the other automata of the SFC program to execute their transitions in a fixed order. This *coordinator* also advances time in an appropriate way. To illustrate the interplay of all automata, consider the SFC given in Fig. 7(a) and the automatically generated timed automata in Fig. 8.



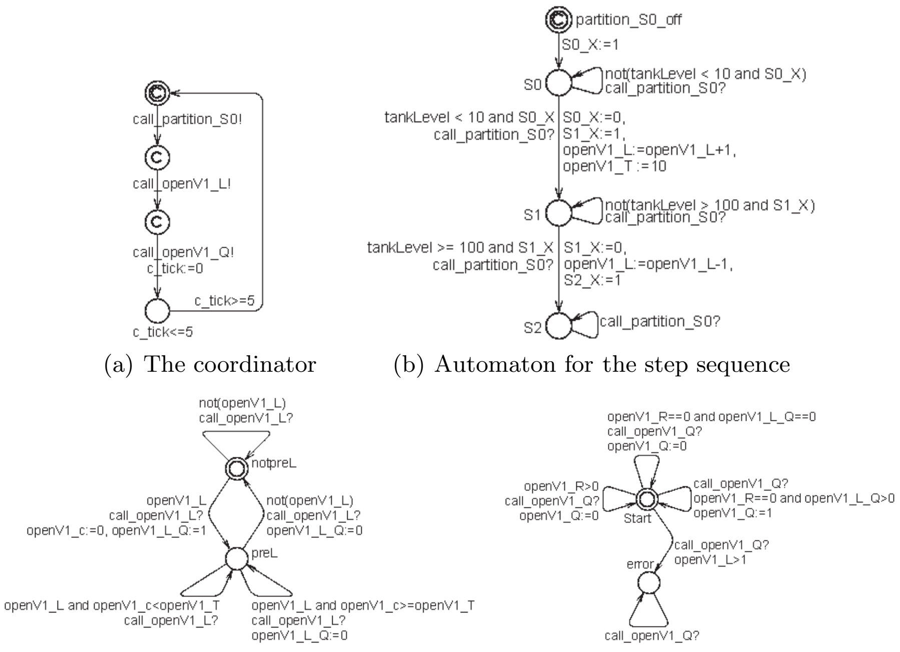
(a) SFC with an action block. (b) SFC with simultaneous sequences.

Fig. 7. Two simple examples of SFC.

The SFC program consists of one simple SFC without simultaneous sequences so that only one automaton is needed to model the sequence of steps and transitions (Fig. 8(b)). The action block attached to the step S1 evokes the action openV1 with the qualifier `L` for a time-limited activity with a duration of 10 sec. For the action openV1 two automata are needed: one for the dynamic behavior of the `L` qualifier of openV1 (Fig. 8(c)) and one to compute the action activity variable openV1_Q (Fig. 8(d)). The automaton shown in Fig. 8(a) coordinates the other three automata in order to emulate the PLC cycle.

Coordinator. The coordinator forces the other components to perform their transitions in a fixed order by the use of binary synchronization channels with the prefix "call_". So-called committed locations, denoted by a "c", are used here to avoid any (non-deterministic) advance of time during the computations. Only in the location of the coordinator that is not marked as committed, the time progresses in order to model the delay between the PLC cycles. The maximum time delay of this location is given by the invariant on the clock variable c_tick that is reset when the location is entered. The minimum delay is given by the guard of the outgoing transition. Thus the delay is deterministic if both are equal.

Simple Step Sequence. Each step and each transition of the SFC has its counterpart in the step sequence automaton, i.e., whenever a transition is taken in the SFC, the corresponding transition in the TA is also taken. Note that the transitions of the automaton can only be taken if the corresponding step activity variables are true; this is important for subordinated SFC. An additional committed location is used to initialize the automaton of the top level SFC graph. Transitions with identical source and target location (self-loops) are necessary



(a) The coordinator (b) Automaton for the step sequence

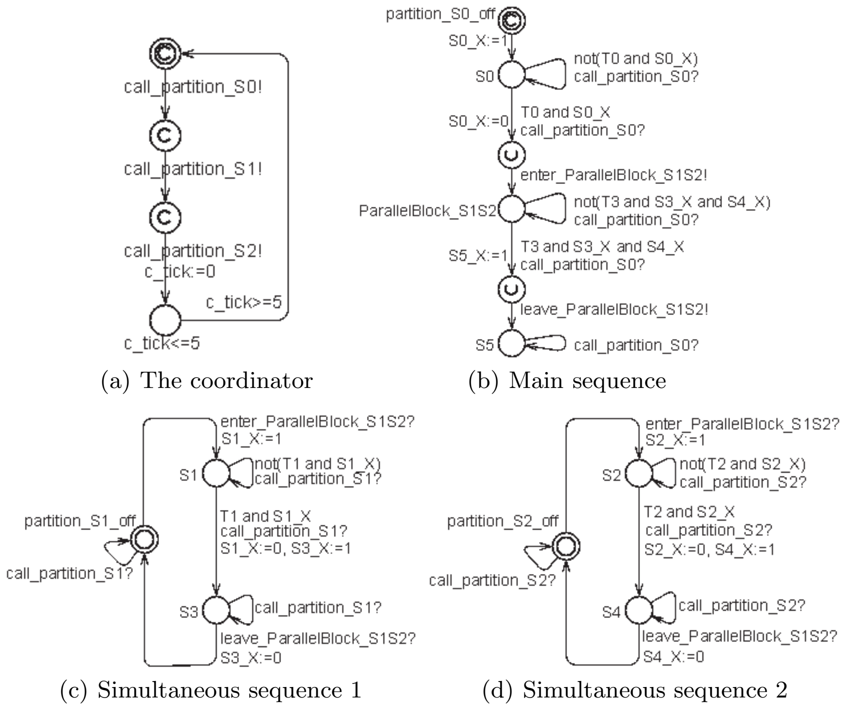
(c) Automaton for qualifier L (d) Computation of action activity

Fig. 8. Automata resulting from the SFC shown in Fig. 7(a).

in order to prevent deadlocks since the coordinator has to synchronize with every other automaton (even if the latter does not change its location). The step activity variables S0_X, S1_X, and S2_X are updated only when the current location changes. The action input variable openV1_L is incremented when the location S1 is entered, and decremented when it is left. An additional variable openV1_T conveys the duration parameter to the action automata.

Action Automata. Fig. 8(c) shows the automaton for the behavior of the `L` qualifier of the action openV1. openV1_c is the clock variable used for the time limiting function. When the integer variable openV1_L becomes greater than zero, the initial location is left, the clock is set to zero, and the output variable openV1_L_Q is set to true. The automaton returns to the initial location only when openV1_L becomes zero again. The output variable openV1_L_Q is set to false when the clock reaches the time limit or when openV1_L becomes zero before the time limit is reached. Fig. 8(d) shows the output automaton for the case that only the qualifier `L` appears in the SFC. The qualifiers `R` (reset) and `N` (not stored) are always included. Note that the reset qualifier has the highest priority. The automaton contains an error location in order to detect forbidden situations defined in the standard, e.g., it is not allowed that steps which reference the same action with a timed qualifier are active at the same time.

Simultaneous Sequences. An SFC with two simultaneous sequences is shown in Fig. 7(b): one consists of the steps S1 and S3 and the transition with the guard T1, and the other consists of S2, S4, and the transition labelled with T2. The main sequence consists of the steps S0 and S5, the transitions guarded by T0 and T3, as well as of a parallel block that encloses the simultaneous sequences. The automata generated for these three sequences are shown in Fig. 9. In the main sequence, the parallel block is represented by one location only, i.e., this location is an abstraction of the simultaneous sequences. The off location of a simultaneous sequence indicates that there is no ready step inside of the sequence. When the parallel block location is entered, the automata of the simultaneous sequences have to leave their off locations and enter the locations S1 and S2. This is achieved by the broadcast channel enter_ParallelBlock_S1S2 which allows the synchronization of one sender with several receivers. Correspondingly, the off locations of the simultaneous sequences are resumed when the parallel block location is left. This can only happen if the steps S3 and S4 are active. The urgent locations (denoted by an "u") before and after the parallel block location



(a) The coordinator (b) Main sequence

(c) Simultaneous sequence 1 (d) Simultaneous sequence 2

Fig. 9. Automata resulting from the SFC shown in Fig. 7(b).

are necessary for the synchronization, since UPPAAL does not support multiple synchronization labels on one transition. Urgent locations have a lower priority than committed locations but are also left instantly.

Hierarchical SFC with History. Subordinated SFC are executed as long as the action they are associated with is active. Hence, it is necessary to deactivate and to activate a subordinated SFC depending on an action activity variable. This is achieved by additional self-loops. Assume that the simple SFC shown in Fig. 10 depends on the action activity variable SFC2_Q. At the beginning, the current location is S1, but the step activity variable S1_X is zero, which means that the step S1 is ready, but not active. The self-loops on the right hand side of the locations model the activation of the corresponding step by setting the step activity variable to one (and incrementing possible action reference variables). The self-loops on the left hand side of the automaton model the deactivation of the SFC by setting the step activity variable to zero (and decrementing possible action request variables). This implementation corresponds to hierarchy with history since the SFC resumes the last step that was activated before. For an implementation without history, all deactivation transitions must lead to the initial location.

4.3 Translation Procedure

We now describe the automatic generation of the UPPAAL model from the SFC.

Translation of Actions. First, all combinations of action names and qualifier symbols that appear in the action blocks of the steps have to be retrieved from the given SFC. For each combination, a corresponding qualifier automaton has to be instantiated (except for the qualifiers `N` and `R`), and an action reference variable is declared. In addition, an action control automaton is created for each action.

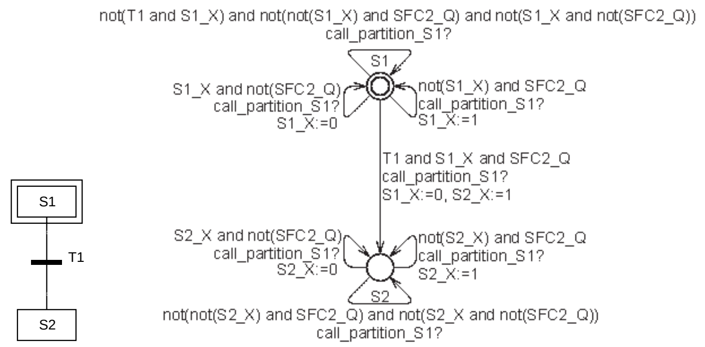


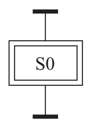
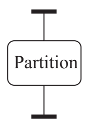
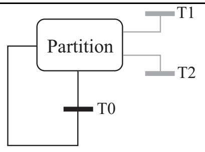
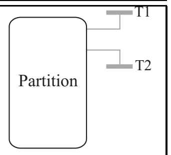
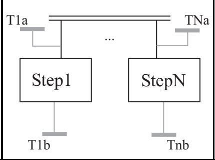
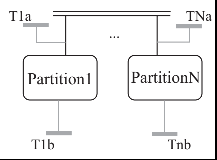
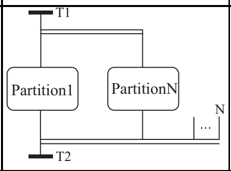
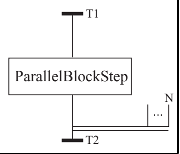
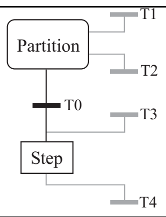
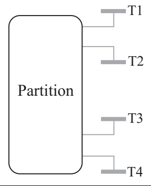
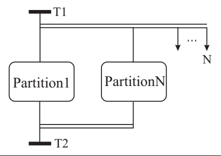
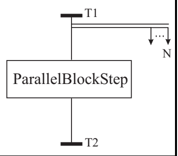
Fig. 10. The SFC associated to the action "SFC2" and the corresponding automaton.

Translation of Charts. The most difficult task in the translation of an SFC program is to identify the simultaneous sequences and to reject malformed charts. A possibility to achieve this in a reliable way even for complex charts containing nested simultaneous sequences (as shown in Fig. 11(a)), entwined loops, and alternative branches is the use of graph grammars [23]. The graph grammar shown in Tab. 1 consists of a set of transformation rules that are applied iteratively to the given chart in order to reduce the graph. The left hand side pattern of a rule defines the situation in which the rule can be applied, and the right hand side gives the result of the transformation. For example, applying the first rule requires an initial step of the chart, and applying the rule replaces the initial by a partition node. In our implementation of the corresponding SFC parser, a rule is always applied to all matching patterns of the graph, before applying another rule. The rules are applied from the first to the sixth, before the procedure continues again with the first one until no further rule can be applied anymore. If the final graph contains only one partition node, the parsing was successful, if not the SFC contains a syntactical error (as, e.g., that two simultaneous sequences lead into a single step).

For illustration, consider the example shown in Fig. 11, and the following sequence of rules:

- rule 1: replace S0 by partition node P0
- rule 2: replace S1 by P1, replace S2 by P2, replace S3 by P3 (→ Fig. 11(b))
- rule 3: P1 takes T1 and S4, P2 takes T2-S5, P3 takes T3-S6 (→ Fig. 11(c))
- rule 6: replace P1 and P2 by PB-Step1 (→ Fig. 11(d))
- rule 2: replace PB-Step1 by partition node P4

Table 1. Parse grammar for SFC graphs.

#:	find pattern:	replace by:	#:	find pattern:	replace by:
1			4		
2			5		
3			6		

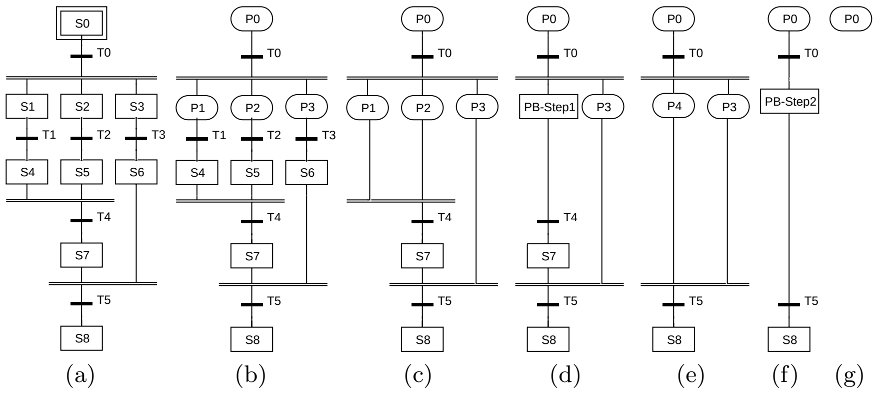


Fig. 11. Successive reduction of a complex SFC graph.

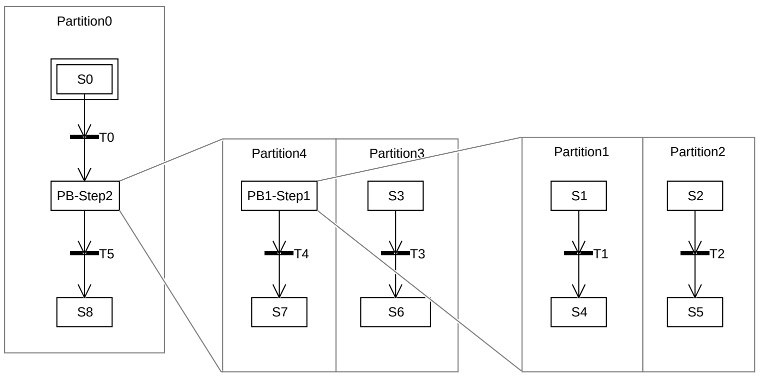


Fig. 12. The identified partitions of the given SFC.

- rule 3: P4 takes T4 and S7 (→ Fig. 11(e))
- rule 5: replace P4 and P3 by PB-Step2 (→ Fig. 11(f))
- rule 3: P0 takes T0 and PB-Step2, P0 takes T5 and S8 (→ Fig. 11(g)).

The partition nodes represent the (nested) simultaneous sequences and the main sequence of the graph. The steps and transitions that belong to a partition are those which are removed by the corresponding transformation step. In the case of a successful transformation, the identified partitions (Fig. 12) are used for the generation of the timed automata: for each partition, a separate automaton is introduced such that the steps and SFC transitions contained in the partition are mapped directly into locations and transitions of the automaton. Depending on whether the SFC graph belongs to the top level or a lower level, and whether the partition contains the initial step or not, different elements such as self-loops, additional locations, etc., are added to the graphs similar to the automata shown in Fig. 9 and Fig. 10.

Generation of the Coordinator. The coordinating automaton simply establishes a loop of steps and transitions that synchronize with each of the other automata in the right order. Only one location is used for modeling the time delay of the PLC cycle. The algorithm that generates all locations and transitions considers the fact that the qualifier automata have to be executed before the corresponding action control automaton. The order of executing the partition automata does not influence the resulting state of the overall model at the end of a cycle.

5 Application to the Evaporation Example

The method described in the previous section is now applied to the evaporation example introduced in Sect. 2. The SFC shown in Fig. 3 realizes the desired operation in the following sense: In the initial step *S0*, the valves *V1*, *V2*, and *V3* are closed and the heater *H1* is switched off by resetting the corresponding Boolean variables. *S0* is left when an input variable *start* is set by an operator, and two parallel branches (starting with *S1* and *Se1*) are activated. In nominal operation (the left branch), the system cycles three times through the sequence from *S1* to *S4*, i.e., *T2* is filled with three batches of *T1*, and is subsequently emptied in step *S5*. If one of the errors is encountered during heating or evaporation, the left branch enters step *Se4*. The right branch leaves *Se1* through one of the transitions labelled by *error1* (heating failure) or *error2* (cooling failure). The actions assigned to *Se2*, or *Se3*, *Se5* and *Se6* respectively, correspond to the exception procedures described above. If the left branch has reached *S6* and the right one *Se7*, the parallel branching is left and the initial state is reached again.

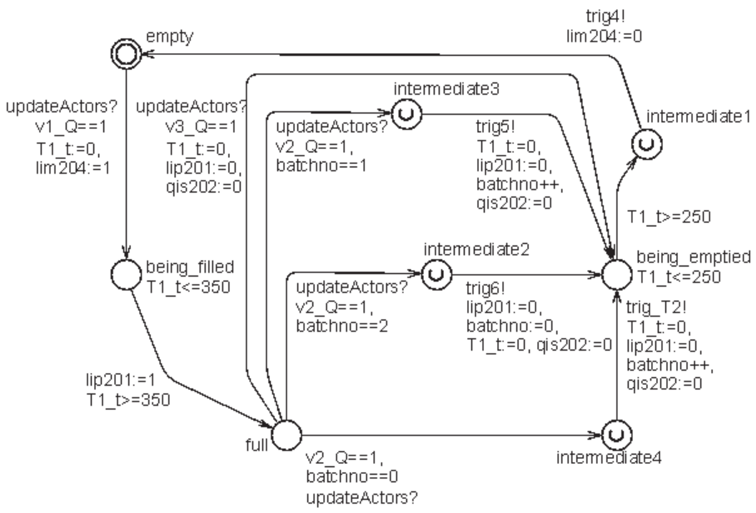


Fig. 13. TA model of the level in tank *T1*.

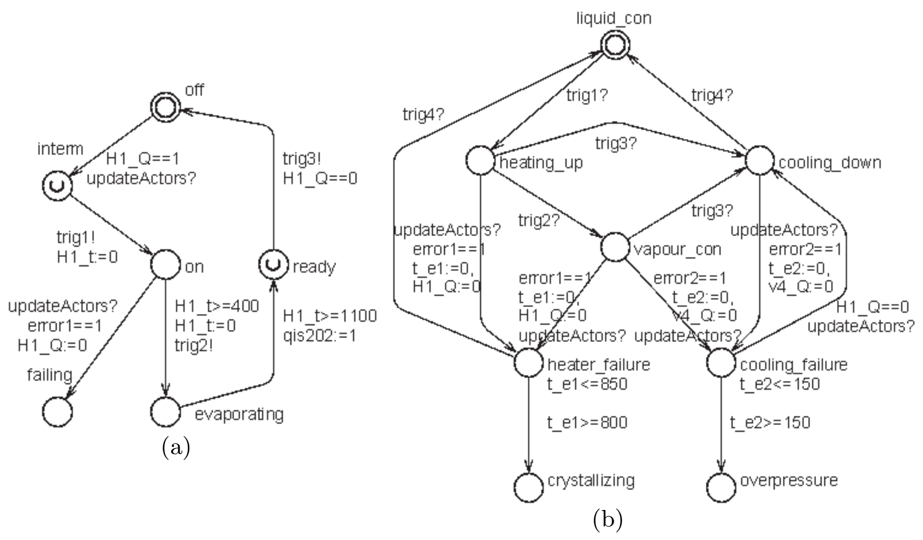


Fig. 14. TA models for the heater *H1* (a) and the state of aggregation of the fluid in *T1* (b).

As mentioned in Sect. 2, the objective of verifying that the temperature in *T1* does never exceed a maximum or minimum value (leading to overpressure or to crystallization in *T1*), requires to consider the plant behavior. We employ a model consisting of one TA each to represent the fluid level of tank *T1*, the level of tank *T2*, the heating effect of the heater *H1*, the state of aggregation of the fluid in *T1*, the mode of operation of the condenser, and an additional automaton that models the occurrence of a failure of the heater or of the condenser. Three of these automata are exemplarily shown in Fig. 13 and 14.

In all cases, the transition times between two events are determined based on measurements for the corresponding laboratory plant, e.g. *T1* is filled in 350 sec (compare to the invariant of the location *being_filled* in Fig. 13). The communication among the plant automata and with the controller automata is realized by synchronization labels and integer variables used in the guard conditions of transitions. The automaton in Fig. 14(b) contains two unsafe plant states, *crystallizing* and *overpressure*, the reachability of which is checked in the verification. It is assumed that the liquid in *T1* crystallizes between 800 and 850 sec after a failure of *H1* occurs, and that the pressure in the evaporator exceeds a critical limit 150 seconds after a cooling failure if heating is continued.

The plant model is composed with the automata generated by the automatic transformation procedure described in Sect. 4. The transformation of the SFC program shown in Fig. 3 yielded 22 automata overall. Three of these correspond to the partitions obtained from applying the rules of the graph grammar. The partition that corresponds to the error handling (the right branch in Fig. 3) is shown in Fig. 15. Another 18 automata represent the action qualifiers, and the

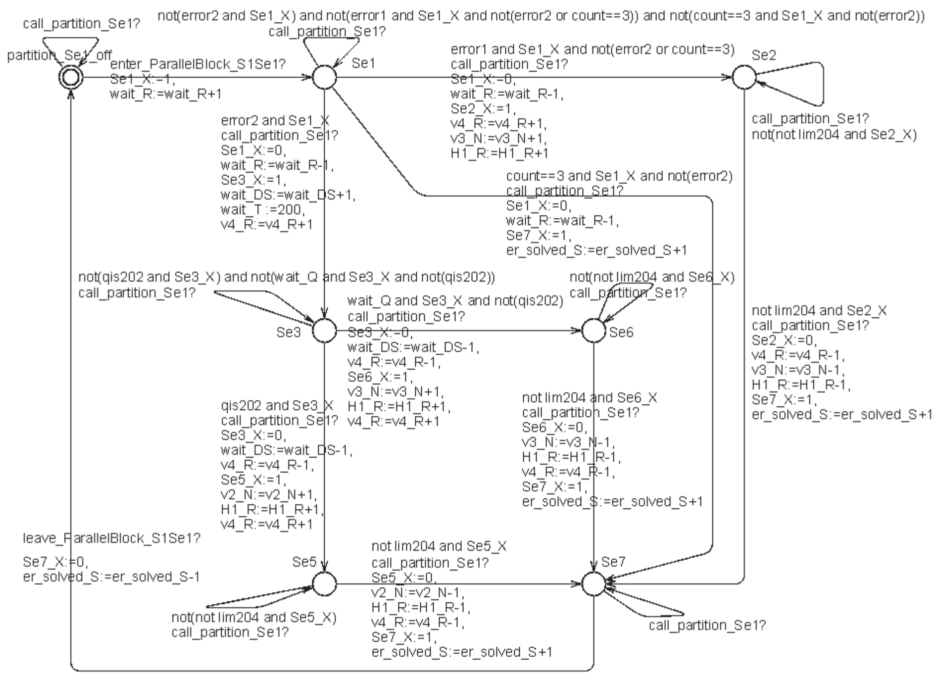


Fig. 15. TA-model of the controller branch for error handling.

model is completed by the coordinator automaton. The relatively large number of automata is a result of the precise emulation of the PLC behavior. However, the overall number of locations is rather small (111), the number of clocks is moderate (10), and the controller part of the model is completely deterministic.

We first verify the safety requirement that the system must never reach the states *crystallizing* and *overpressure* (according to requirement (b) in Sect. 2). On a PC with a 1000 MHz-Duron processor, the analysis with Uppaal (version 3.4) terminates after less than 10 seconds with the result that both states are not reachable, i.e. the controller is designed correctly with respect to the safety requirement. To verify that the controller realizes the desired production cycle (requirement (a)), we analyze for the plant automaton of the tank *T2* whether the state *emptied* is reachable after the tank was filled. For illustration, a relatively large PLC cycle time of 50 sec was chosen for these experiments. This analysis terminates after approx. 15 minutes with the result that the state is reachable. The difference in computation time for both experiments is due to the different length of the traces that lead to the final states for the two requirements.

6 Conclusions

The benefits of the approach presented here are (i) it starts from a controller representation that is a de-facto standard for specifying PLC programs in industry, and (ii) the procedure is completely algorithmic once the SFC controller, the formal requirements, and the plant model are available. The steps of designing the SFC controller, obtaining the plant model, and correcting the design in case of a negative verification result, can obviously not be accomplished completely algorithmically. Our current aim is, however, to develop a scheme to derive the SFC program systematically from the set of specifications (usually given in natural language).

When applying the approach described here, an important issue is to employ a plant model of sufficient accuracy. At least for chemical processing systems, we have experienced that FSA models are often not sufficient to verify the exclusion of safety-relevant plant states. The use of TA models is appropriate if the transition times between certain events can be estimated (or measured) accurately and conservatively. If this is not possible, one can start from hybrid dynamical models and derive TA algorithmically, e.g., by the procedures described in [24].

The choice of the plant model but also the level of detail of the controller model determines the complexity of the verification task. A system of the size of the example in Sect. 5 can be verified in a few minutes. It should be mentioned, however, that the main complexity here arises from the fact that the PLC cycle is mapped into the TA model. To reduce this source of complexity we currently investigate how to separate the verification of requirements for which the cyclic operation is relevant from the analysis of those for which it is not.

Acknowledgments

This work was financially supported by the German Research Council (DFG) within the priority program *Integration of Software Specification Techniques for Applications in Engineering* under the grants EN 152/32-2 and RO 1122/10-2. Nanette Bauer contributed to this work during her time at the Process Control Lab, University of Dortmund. National ICT Australia is funded through the Australian Government's *Backing Australia's Ability initiative,* in part through the Australian Research Council.

References

1. Lewis, R.: Programming industrial control systems using IEC 61131-3. IEE (1998)
2. International Electrotechnical Commission, Technical Committee No. 65: Programmable Controllers – Programming Languages, IEC 61131-3. (2003) Ed. 2.0.
3. Moon, I., Powers, G.J., Burch, J.R., Clarke, E.M.: Automatic verification of sequential control systems using temporal logic. AIChE Journal **38** (1992) 67–75
4. Kowalewski, S., Engell, S., Preussig, J., Stursberg, O.: Verification of logic controllers for continuous plants using timed condition/event system models. Automatica **35** (1999) 505–518

5. Lampérière, S., Lesage, J.J.: Formal verification of the sequential part of PLC programs. In Boel, R., Stremersch, G., eds.: Discrete Event Systems, Kluwer Academic Publishers (2000) 247–254
6. Bauer, N., Huuck, R.: Towards automatic verification of embedded control software. In: Proc. 2nd Asia-Pacific Conf. on Quality Software. (2001) 561–567
7. Clarke, E., Wing, J.: Formal methods: State of the art and future directions. ACM Computing Surveys **28** (1996) 626–643
8. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)
9. Fujino, K., Imafuku, K., Yamashita, Y., Nishitani, H.: Design and verification of the SFC program for sequential control. Comp. Chem. Eng. **24** (2000) 303 – 308
10. L'Her, P., Scharbarg, J., Le Parc, P., Marce, L.: Proving sequential function chart programs using automata. LNCS **1660** (1998) 149 – 163
11. Bornot, S., Huuck, R., Lakhnech, Y., Lukoschus, B.: Verification of sequential function charts using SMV. In: Proc. Int. Conf. on Parallel and Distributed Processing Techniques and Applications. (2000) 2987–2993
12. McMillan, K.: The SMV Language. Cadence Berkeley Labs. (1999) http://www-cad.eecs.berkeley.edu/kenmcmil/language.ps.
13. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.: Symbolic model checking: 10^{20} states and beyond. Information and Comp. **98** (1992) 142–170
14. Havelund, K., Larsen, K., Skou, A.: Formal verification of a power controller using the real-time model checker Uppaal2k. In: Proc. 5th AMAST Workshop, ARTS'99. (1999) 277–298
15. Bauer, N., Huuck, R., Lukoschus, B., Engell, S.: A unifying semantics for sequential function charts. In: This volume. (2004)
16. Clarke, E., Emerson, E.: Synthesis of synchronisation skeletons for branching time temporal logic. In Kozen, D., ed.: Workshop on Logic of Programs. Volume 131 of LNCS., Springer-Verlag (1982) 52–71
17. Queille, J., Sifakis, J.: Specification and verification of concurrent systems in CESAR. In: 5th Int. Symp. on Progr. Volume 137 of LNCS., Springer-Verlag (1982) 337–350
18. Bauer, N., Kowalewski, S., Sand, G., Löhl, T.: A case study: Multi product batch plant for the demonstration of control and scheduling problems. In: Proc. Analysis and Design of Mixed Processes. (2000) 383–388
19. Larsen, K.G., Pettersson, P., Yi, W.: Compositional and Symbolic Model-Checking of Real-Time Systems. In: Proc. of the 16*th* IEEE Real-Time Systems Symposium, IEEE Computer Society Press (1995) 76–87
20. Behrmann, G., David, A., Larsen, K.G., Mller, O., Pettersson, P., Yi, W.: Uppaal – present and future. In: Proc. of 40*th* IEEE Conference on Decision and Control, IEEE Computer Society Press (2001)
21. Larsen, K.G., Pettersson, P., Yi, W.: Uppaal in a Nutshell. Int. Journal on Software Tools for Technology Transfer **1** (1997) 134–152
22. Aceto, L., Bergueno, A., Larsen, K.G.: Model Checking via Reachability Testing for Timed Automata. In: Proc. 4*th* Int. Workshop on Tools and Algo. for the Constr. and Analysis of Systems. Volume 1384 of LNCS., Springer-Verlag (1995) 263–280
23. Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G.: Handbook of graph grammars and computing by graph transformation: vol. 2: applications, languages, and tools. World Scientific Publishing Co., Inc. (1999)
24. Stursberg, O.: Analysis of switched continuous systems based on discretization. In: Proc. 4th Int. Conf. on Automation of Mixed Processes. (2000) 73–78