A Model-Checking Approach to Safe SFCs

Ralf Huuck
Department of Computer Science
University of Kiel, Germany
Email: rhu@informatik.uni-kiel.de

Ben Lukoschus
Department of Computer Science
University of Kiel, Germany
Email: bls@informatik.uni-kiel.de

Nanette Bauer
Department of Chemical Engineering
University of Dortmund, Germany
Now with BASF AG, Ludwigshafen, Germany

***Abstract*— Sequential function charts (SFC) are a high-level graphical programming language for programmable logic controllers. Their main purpose is to provide a structure and organization of the control flow. Therefore, various features such as parallelism, priorities on branching transitions, and activity manipulations are incorporated. The syntactic rules for building SFCs are formally defined in IEC 61131-3. It is, however, still possible to derive SFCs from these rules whose structure do not make sense. In this work we give a characterization for so-called *safe SFCs*. Moreover, we present a semantic definition for them, as well as an algorithmic approach to automatically detect whether an SFC is safe or not.**

I. Introduction

Computers penetrate nearly every area of our daily life. They control telephone systems, automated teller machines, cars, etc. Everything is computer driven or at least supported by computers. But a major part of the computer systems we never see. All the things natural to us like water, food, and electricity rely on computers as far as their distribution and manufacturing process is concerned.

Wherever we get in touch, directly or indirectly, with computers and computer driven hardware, their main purpose is: control. The control of the traction of the car wheels, of restricted access, the scheduling and transport along the assembly lines for food packaging, and the control of many safety-critical applications such as fission control in nuclear power plants.

The systems responsible for control are mostly not general-purpose PCs but specific industrial computers. The reasons are simple and stem from the requirements of such control systems: they have to be robust, reliable and cheap. Robustness to ensure that these systems also work under hazardous conditions, which might be heat, dust, and electro-magnetic noise [1]. Reliability in order to have them running 24 hours a day over 5 or even 10 years. And they have to be cheap to enter a mass market and to reduce the costs of the manufacturing process.

Next to micro-controllers, specific purpose computers on a single chip, a prominent class of industrial controllers are so called *programmable logic controllers* (PLC). First developed during the early 1970s, PLCs started as simple devices to replace electro-mechanical relays. Using integrated circuit technology, they performed simple sequential control tasks, in isolation from other control and monitoring equipment. These simple devices have grown into complex systems capable of almost any type of control application, including motion control, data manipulation, and advanced computing functions. Nowadays, PLCs are extensively used in the field of automation, and they are integrated into much larger environments, requiring communication with other controllers or computer equipment for plant management functions [2], [3].

Control systems driven by PLCs are often complex, safety critical, and involve a lot of money. Any failure of these systems might not only result in a significant financial loss but can lead to casualties as well. Hence, next to robustness and reliability, their actual programming and the correctness of the programs plays a vital role.

There exist different programming languages for PLCs. These have been designed with an emphasis on control tasks, each intended for a specific application domain, and based on the background of the control engineers who use them. The standard IEC 61131-3 was developed to achieve more conformity of the different PLC programming languages.

One of the languages defined in this standard is called *Sequential Function Charts (SFCs).* This is a graphical, high-level programming language which aims at providing a clear understanding of the possibly interwoven program parts. SFCs allow to decompose and structure program parts and include interesting concepts such as parallelism, activity manipulation and hierarchy.

The standard provides formal rules for building SFCs, i.e., there is a well-defined syntactic framework. However, not every SFC built according to these rules makes sense. The structure of an SFC may yield that the SFC is "unsafe" or "unreachable", which are notions defined in the standard. Such SFC structures are declared as erroneous. The standard, however, provides neither means to detect such structures nor any precise characterization of this phenomenon.

In this work we pinpoint the characteristics for such erroneous structures and, moreover, provide means to detect them. This detection is based an on algorithmic approach and as such can take place fully automatically.

The remainder of this work is organized as follows: In Section II we present a brief introduction to SFCs, followed by a characterization and formal definition of safe SFCs in Section III. The subsequent Section IV introduces simple SFCs, an abstraction of SFCs which is sufficient for a decision procedure for safe SFCs. In Section V we define an algorithmic approach to check for safe SFCs. This decision procedure relies on model checking simple SFCs for particular requirements. Therefore, this section includes a translation

from simple SFCs to the input language of a model checker, as well as the automatic generation of the according safety requirements. All this is illustrated by an example in Section VI. The conclusions and future work are discussed in Section VII.

II. Sequential Function Charts

Sequential function charts are defined in [4] as elements of a graphical programming and structuring language for programmable logic controllers. The SFC definitions in the standard IEC 61131-3 are based on IEC 60848 [5], which defines the specification language Grafcet. Grafcet in turn is strongly related to Petri nets [6].

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

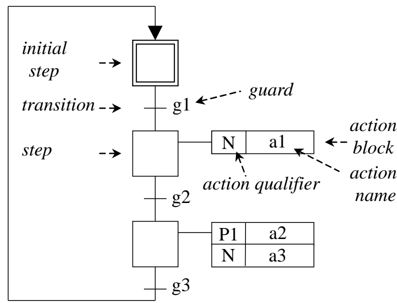

Fig. 1. Elements of SFCs

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 duration of the respective action (e.g., permanent or limited in time), and the *action name,* which identifies the program that is run when the action is active.

An SFC does not necessarily have to be a single sequence of steps and transitions. For SFCs we can identify a number of different transition types (cf. Fig. 2):

- *simple transitions* between two steps as denoted in (1),
- *alternative branching*, i.e., the choice between several transitions as in (2),
- *divergence* as in (3), i.e., a parallel branching from one step into a set of parallel steps, and
- *convergence* (4) which denotes a synchronization of several parallel steps into a single step.

Furthermore, direct combinations of both convergence and divergence as depicted in (5) are allowed. Moreover, the standard refers to transitions like the one guarded by g_3 in Fig. 1 as *loops*. From our point of view loops as well as alternative branching belong to the same class, namely (sets of)

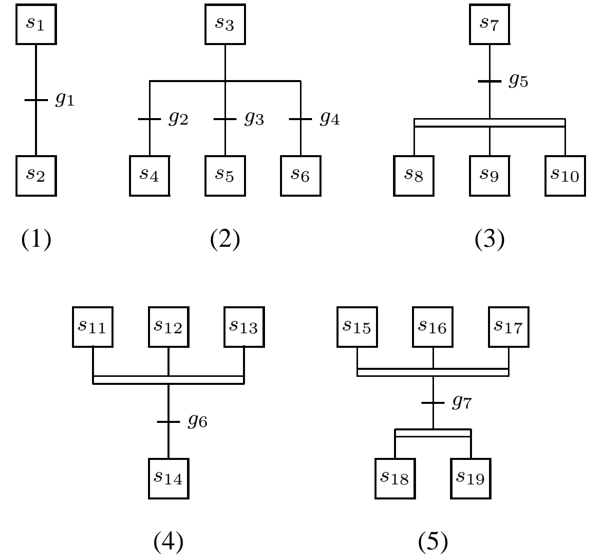

Fig. 2. Basic transition types

simple transitions. Also note that any transition drawn without an arrowhead is understood as going from top to bottom.

An additional feature of SFCs is to explicitly assign transition priorities to alternative branches. When firing transitions the one which has the highest priority among the enabled ones will be taken.

An SFC, like any PLC program, operates in a cyclic mode. At the beginning of each cycle, inputs are read from the environment (e.g., sensor readings) and stored in variables. Then all programs associated with active actions are executed, resulting in a computation that changes some variables. After that, all guards on transitions starting at active steps are evaluated, and the enabled transitions are taken, changing the set of active steps. At the end of the cycle, outputs (which usually depend on the previous computation) are sent to the environment. Then the next cycle starts. A full SFC semantics for a comprehensive class of SFCs can be found in [7].

The basic transition types shown in Fig. 2 can be combined into more complex transition structures like those shown in Fig. 1 and Fig. 3. However, there are various combinations which do not seem to make sense, e.g., in the SFC shown in Fig. 3. There we have the transition from s_4 to s_1 which jumps out of a parallel branch, the transition from s_2 to s_5, which jumps from one simultaneous branch to another, and the transition from s_5 and s_6 to s_7, which is a convergence of simultaneous sequences of two steps which are part of an alternative branching starting in step s_3.

Although the standard forbids to use such ill-structured SFCs (which are called “unsafe” or “unreachable”, see Section 2.6.5 of [4]), it does not give a precise characterization of this phenomenon. In the following we define the notion of *safe SFCs* which subsumes the absence of “unsafe” and “unreachable” structures and give a semantic characterization. Moreover, we define an algorithmic approach to determine automatically for any given SFC if it is safe or not.

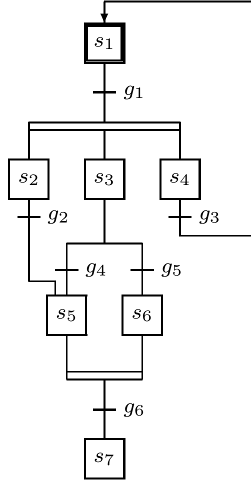

Fig. 3. "Unsafe" SFC

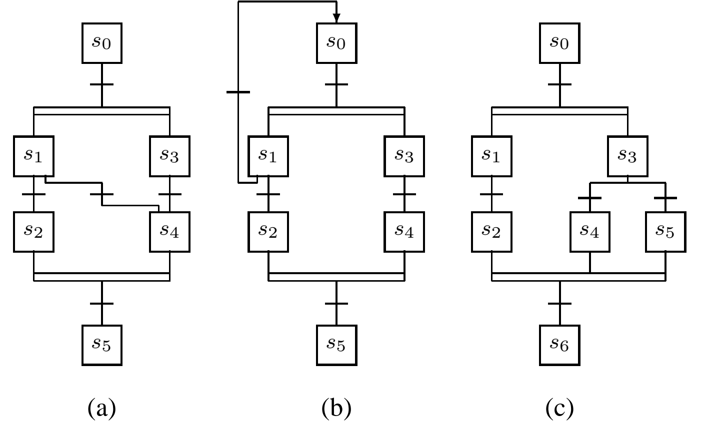

Fig. 4. Various types of "unsafe" and "unreachable" SFCs

III. Safe SFCs

The structure of the SFC depicted in Fig. 3 complies to the syntax as given in IEC 61131-3. The SFC as such, however, does not make sense. We like to stress that this observation does not depend on the semantics of the given guards or any possible associated actions. It is a structural problem in the first place.

We characterize what kind of constructions do not make any sense. Any SFC without a divergence/convergence is always sensible as long as the syntactic rules are followed. However, if we allow parallelism there are three major violations that might occur (cf. Fig. 4):

1) There is a jump between different parallel branches without a proper synchronization first (SFC a). This violates a proper parallel structure.
2) There is a jump out of a parallel branch, probably into a different parallel branch (SFC b). This violates a proper synchronization.
3) Two or more alternative branches of the same parallel branch are synchronized (SFC c). This does not make sense either.

This leads to the following characterization:

Characterization 1 (Safe SFC)
An SFC is safe *if and only if there are no jumps between parallel branches, no jumps out of parallel branches, and every branch is properly synchronized.*

Although we have characterized safe SFCs by the above definition, we have not yet explored the semantic consequences. This is done next.

The first two items of the characterization above describe a breach of proper parallelism. Let us consider parallel branches as independent *processes:* Whenever a diverging transition is taken, in each of the target steps a process is started, and it is terminated when the respective converging transition is taken. We clearly want control not to jump from one process to another when in fact we demand a proper synchronization.

Semantically, this means that control might reside simultaneously at different places in the same process. To understand this, imagine the following scenario in Fig. 4 (a): Control starts from step s_0 and moves on to s_1 and s_3. While in the right branch control remains in s_3 it moves from the left in s_1 to the right in s_4. Now, control resides in two steps in the same process, i.e., in s_3 and s_4. In SFC (b) we can construct a similar scenario with the same result.

If we consider control as *tokens* (as in a Petri net setting) moving around according to the SFC transition structure, we can phrase one condition for a safe SFC as follows: There is never more than one token simultaneously in the same process. If we completely abstract an SFC from guards and actions and allow control to reside in a location for an arbitrary number of cycles, we can rephrase the condition as: Determine that no step ever has more than one token. The abstraction to SFCs without guards where control may reside in a step for an arbitrary time is a sensible abstraction, since we want to verify structural properties that do not depend on specific guards and actions.

The issue of proper synchronization is slightly different. The SFC (c) in Fig. 4 is not "unsafe", since control cannot simultaneously reside in two processes. But it is "unreachable", since there are not enough tokens for the convergence, which is of course due to the alternative branching. Hence, the second requirement is that there has to be a possibility to simultaneously reach all source steps of any converging transition. Again it is useful to consider SFCs abstracted from any guards and actions.

Now we can formally define our requirements.

Definition 1 (Safe SFC)
An SFC is safe *iff*

1) *for all possible executions there is at most one token in a process and*
2) *for any converging transition there exists an execution such that the transition can be taken.*

Both requirements can be stated as reachability problems over finite graphs and, hence, are solvable by model checking. In the following we describe a simple model-checking framework for SFCs where the respective requirements can be checked automatically.

IV. A Simplified Formal Model

In order to model-check SFCs, we first give a formal SFC model. It serves as a basis for the latter translation into the input language of a model checker. Since we are only interested in structural properties of SFCs, there is no need to define SFCs with their full set of features. In particular, we are only interested in the overall SFC structure, i.e., the steps and transitions. We call such a reduced model a *Simple SFC*. We define the syntax for a Simple SFC as follows:

Definition 2 (Simple SFC)
A Simple SFC consists of a triple $\mathcal{S} = (S, s_0, T)$, where

- *S is a finite set of steps,*
- *$s_0 \in S$ is the initial step, and*
- *$T \subseteq (2^S \setminus \{\emptyset\}) \times G \times (2^S \setminus \{\emptyset\})$ is a set of transitions, each transition labeled with a guard g from a set G of transition conditions.*

For every transition $(S_s, g, S_t) \in T$ the set S_s is the non-empty set of source steps and S_t the non-empty set of target steps. For simple transitions both sets are singletons. Alternative branching is modeled through a set of simple transitions, i.e., several transitions from the same source step to different target steps. For divergence transitions the set of target steps S_t is a non-singleton set and for convergence S_s is a non-singleton set. Note that this simple model does not include priorities, actions and hierarchy. It is, however, sufficient for deciding safe SFCs, since they do not depend on possible priorities or the semantics of actions. Hierarchical SFCs can be checked compositionally, i.e., each SFC is checked independently and, as such, they can be modeled independently.

Briefly, the evolution of a Simple SFC is defined as follows: Control starts in the initial step. Whenever a guard g evaluates to $true$ and control resides in all source steps the respective transition, it moves on to all target steps of that transitions.

V. Model Checking for Safe SFCs

Model checking [8], [9] is a formal verification technique based on an automatic, exhaustive enumeration of all possible behaviors of a system. Provided we have a formal execution model M of a system (the operational semantics for SFCs in our case) and some property φ (we are interested in checking Definition 1 for a given SFC), a model checker can be used to compute the validity of φ in M, often stated as $M \models \varphi$.

In the following we define a translation from Simple SFCs to the input language of a model checker in order to check for safe SFCs, i.e., compliance with Definition 1. Although any general-purpose model checker can be used for this task, we define the translation and verification process for the *Cadence SMV (CaSMV)* model checker [10].

A. Translation of Simple SFCs to CaSMV Models

CaSMV is a symbolic model checker which supports the verification of temporal logic properties of Kripke structures [11]. 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, written as `q` and `next(q)` in CaSMV notation. In order to translate a Simple SFC $\mathcal{S} = (S, s_0, T)$ to CaSMV we mimic the transition relation as given by the Simple SFCs. Therefore, we introduce the following Boolean variables in our CaSMV model of $\mathcal{S}$:

- $\mathtt{s}_i$ for each $s_i \in S$, i.e., one variable for each step of the SFC. These variables model whether there is a token in this step, i.e., control resides in it. Initially, $\mathtt{s}_0 = true$, and for all $i \neq 0$, $\mathtt{s}_i = false$.
- $\mathtt{g}_i$ for each g_i. This variable is $true$ whenever its associated transition condition evaluates to $true$. We do not put any restrictions on the initial value and the changes of any $\mathtt{g}_i$, and thus over-approximate the real SFC behavior, in which the evaluation of a guard usually depends on variables not contained in our CaSMV model.

We map the transition relation of a Simple SFC to CaSMV by explicitly defining the `next`-state of all step variables, i.e., we define how state changes evolve over time.

The `next`-state of a step variable $\mathtt{s}_i$ of a step s_i has to be $true$, i.e., there is a token in it, if and only if there is a transition taken into s_i in the `next`-state or it is already $true$ now and there is no transition taken that leaves s_i. To put it simple, if a token is moved to s_i the step variable $\mathtt{s}_i$ becomes $true$ and it remains $true$ until the token is moved away from s_i.

Formally this can be expressed by:

$$\mathtt{next}(\mathtt{s}_i) \equiv \mathtt{s}_i_\mathit{will_be_entered} \vee (\mathtt{s}_i \wedge \mathtt{s}_i_\mathit{will_not_be_left}).$$

If s_i is entered depends on two conditions. First, there has to be a transition t that is enabled in the next cycle and that contains s_i in its target set. Second, any other transition t' with the same source set is not taken. The second part is necessary to cope with alternative branches when several guards are $true$ at the same time. E.g., when taking a transition leaving s_3 in Fig. 2, at most one of s_4, s_5, s_6 can be active in the next cycle.

$$\begin{aligned} &\mathtt{s}_i_\mathit{will_be_entered} \equiv \\ &\quad (\exists t = (A, g, B) \in T \,:\, s_i \in B \wedge \mathtt{next}(\mathtt{g}) \wedge \bigwedge_{s_j \in A} \mathtt{s}_j) \\ &\wedge (\forall t' = (A, g', B') \in T \setminus \{t\} \,: \\ &\quad \mathtt{next}(\mathtt{g}') \Rightarrow \bigwedge_{s_j \in B' \setminus B} \neg \mathtt{next}(\mathtt{s}_j)) \end{aligned}$$

If s_i already has a token, $\mathtt{s}_i$ remains $true$ if no transition having s_i in its source set is enabled:

$$\begin{aligned} &\mathtt{s}_i_\mathit{will_not_be_left} \equiv \\ &\quad \neg\exists(A, g, B) \in T \,:\, s_i \in A \wedge \mathtt{next}(\mathtt{g}) \wedge \bigwedge_{s_j \in A} \mathtt{s}_j \end{aligned}$$

Generally, the evolution of the guards have to be specified according to the information they reason about, e.g., input variables which correspond to a PLC's sensor data. In our setting, however, the structural properties are independent of any specific guards. Thus, we chose one distinct Boolean variable for each guard at each transition and leave this variable unspecified. As a result the guards evolve non-deterministically, which just reflects all arbitrary behaviors independent of specific semantic issues.

B. Generating Verification Conditions

Having defined a translation from Simple SFCs to the input language of the model checker, it remains to translate the requirements for safe SFCs, as given in Definition 1, into verification conditions. This is done separately for each of the two requirements.

1) At most one token in a process: In order to check that there is never more than one token in a process, it suffices, due to the non-determinism, to check that there is never more than one token in a step. For technical reasons we distinguish two cases where such a situation can arise: Firstly, if a token resides in a step and another token moves into this step and, secondly, if there are two tokens from different transitions moving into the same target step.

In preparation to check that there might be more than one token in a step at a time we introduce an additional variable `token_overflow` to indicate just this. We define `token_overflow` such that it is set to *true* whenever there is a token in a step that can be entered by another transition or there are two transitions that can enter the same step simultaneously. Initially, the variable is set to *false*.

Formally, the evolution of `token_overflow` checks the two conditions above for all steps $s_i \in S$:

$$\begin{aligned}&\texttt{next(token_overflow)} \equiv \bigvee_{s_i \in S} \Big(\\ &\quad (\texttt{s}_i \wedge \bigvee_{(A,g,B)\in T \wedge A \neq B} (s_i \in B \wedge \texttt{next(g)} \wedge \bigwedge_{s_j \in A} \texttt{s}_j)) \\ &\vee (\bigvee_{(A_1,g_1,B_1)\in T \wedge (A_2,g_2,B_2)\in T \wedge A_1 \cap A_2 = \emptyset} (s_i \in B_1 \cap B_2 \wedge \\ &\quad \texttt{next(g1)} \wedge \texttt{next(g2)} \wedge \bigwedge_{s_j \in A_1 \cup A_2} \texttt{s}_j))\Big)\end{aligned}$$

The first condition states that we have a possible token overflow in s_i whenever s_i and all source steps of an enabled transition leading to s_i have a token. The reason is, in the next cycle the tokens from the source step might move to s_i, while the token in s_i can remain in s_i due to the non-deterministic guards. This results into more than one token in s_i. Note that we need $A \neq B$ to exclude self-loops from the test above, since the occurrence of a token leaving and entering the same step simultaneously is not considered to be an error.

The second condition checks if there are two transitions with different source steps ($A_1 \cap A_2 = \emptyset$) having s_i is in both their target sets. If in the next cycle both transitions are enabled, then both tokens can move into the same step s_i.

Having defined this, the verification condition for the requirement that there is at most one token in a process boils down to the requirement that there is no token overflow. Stated as a temporal logic formula in CaSMV syntax:

```
SPEC AG !token_overflow.
```

This means that in *all* executions at *any* time a token overflow does not occur.

2) Proper synchronization: The verification property for proper synchronization is slightly different. We have to check for every converging transition $(A, g, B) \in T$ with $|A| > 1$ that there is a least one execution in which at some point all steps in A have a token. We define T_c as set of all converging transitions:

$$T_c = \{(A, g, B) \in T \mid |A| > 1\}.$$

We require for all $(A, g, B) \in T_c$:

$$\texttt{SPEC EF } \&_{s_i \in A} \texttt{s}_i \,.$$

This means for all converging transitions that eventually there is a token in all source steps. Note, however, that this needs not to be the case for the concrete SFC. Here, we abstract from any restrictions imposed by guards etc. and, therefore, over-approximate the behavior to test for the structural compliance.

The whole translation process as well as the generation of the verification conditions and their checking can be done fully automatic. This means, there is no user interaction required in testing for safe SFCs.

C. Reduction of Model Size

One disadvantage of the previous translation and verification process is that we explicitly model each transition condition by a Boolean variable. However, the verification is independent of these conditions, i.e., we do the check for every possible combination of guard evaluations. If it is possible to perform the same checks without using a variable for each guard, we can reduce the potential state space of the model significantly. Such an approach is introduced next.

We suggest the following: We substitute in the translation process every guard by the constant *true*. This allows control to move freely through the SFC. However, this alone is not sufficient. By the synchronous execution mechanism in each cycle every enabled transition has to be taken. This means in particular, if all guards evaluate to *true* control has to move out of a step in every cycle (if possible). However, we like the control to behave arbitrarily, in particular it should be possible to remain in a step.

Therefore, we rephrase our definitions for the transition relation in such a way that guard variables are no longer used in the CaSMV model. Instead of using guard variables to determine if a transition is taken, we use the existence of token in its source or target steps after the transition.

$$\begin{aligned}&\texttt{next(s}_i\texttt{)} \equiv \\ &\quad \texttt{s}_i\textit{_will_be_entered}' \vee (\texttt{s}_i \wedge \texttt{s}_i\textit{_will_not_be_left}')\end{aligned}$$

where

$$\mathtt{s}_i\text{_}\mathit{will_be_entered}' \equiv \\ (\exists t = (A, g, B) \in T\ : \\ s_i \in B \wedge \bigwedge_{s_j \in A} \mathtt{s}_j \wedge \bigwedge_{s_k \in A \setminus B} \neg\mathtt{next(s}_k\mathtt{))} \\ \wedge (\forall t' = (A, g', B') \in T \setminus \{t\}\ : \bigwedge_{s_j \in B' \setminus B} \neg\mathtt{next(s}_j\mathtt{))}$$

and

$$\mathtt{s}_i\text{_}\mathit{will_not_be_left}' \equiv \\ \neg\exists(A, g, B) \in T\ : s_i \in A \wedge \bigwedge_{s_j \in A} \mathtt{s}_j \wedge \bigwedge_{s_k \in B} \mathtt{next(s}_k\mathtt{)}$$

We also remove guard variables from the definition of `token_overflow`:

$$\mathtt{next(token_overflow)} \equiv \bigvee_{s_i \in S} (\\ (\mathtt{s}_i \wedge \bigvee_{(A,g,B) \in T \wedge A \neq B} (s_i \in B \wedge \bigwedge_{s_j \in A} \mathtt{s}_j)) \\ \vee (\bigvee_{(A_1,g_1,B_1) \in T \wedge (A_2,g_2,B_2) \in T \wedge A_1 \cap A_2 = \emptyset} (s_i \in B_1 \cap B_2 \wedge \\ \wedge \bigwedge_{s_j \in A_1 \cup A_2} \mathtt{s}_j)))$$

Here it is sufficient just to remove all guard variables (more precise: replace them with $true$), since whenever the variable `token_overflow` becomes $true$ in the old definition, the guard variables occurring in the formula are $true$ anyway.

VI. EXAMPLE

Let us consider the SFC of Fig. 3. The transition relation in CaSMV as defined in Section V-C is given in Fig. 5. As you can observe, a step s_i obtains a token, i.e., becomes $true$ (`1` in CaSMV), if there is a token in the source step of a transition to s_i which is removed in the next cycle and is not moved to some other step than s_i. A token is removed from a step s_i (s_i is set to `0`), if there is a transition t leaving s_i such that all source steps of t have tokens now and all target steps in the next cycle. Not shown in this figure is that initially all steps but the initial step s_1 are $false$, i.e., there is only one token in the initial step.

The CaSMV code for the `token_overflow` variable according to the definition in Section V-C is presented in Fig. 6. It shows that the `token_overflow` variable is set to $true$ whenever there is a token in a step and another transition into this step or if there are two steps with a token which have the same target step.

The verification conditions are shown in Fig. 7. They state that there should be no token overflow and the transition involving s_5 and s_6 have to properly synchronize.

The first verification condition, however, is violated. Hence, the SFC is not safe. Moreover, a counter example is produced by the model checker that helps to improve the structure such that the SFC becomes safe. Note that the second verification condition only holds because of the additional unsafe structure

```
default next(s1) := s1;
 in case{
     s4 & ~next(s4)                          :=1;
     s1 & next(s2) & next(s3) & next(s4) :=0;}

default next(s2) := s2;
 in case{
     s1 & ~next(s1)              :=1;
     s2 & next(s5)               :=0;}

default next(s3) := s3;
 in case{
     s1 & ~next(s1)              :=1;
     s3 & (next(s5) | next(s6))  :=0;}

default next(s4) := s4;
 in case{
     s1 & ~next(s1)              :=1;
     s4 &  next(s1)              :=0;}

default next(s5) := s5;
 in case{
     s2 & ~next(s2)  :=1;
     s3 & ~next(s3) & ~next(s6)  :=1;
     s5 &  s6 & next(s7)         :=0;}

default next(s6) := s6;
 in case{
     s3 & ~next(s3) & ~next(s5)  :=1;
     s5 &  s6 & next(s7)         :=0;}

default next(s7) := s7;
 in case{
     s5 & s6 & ~next(s5) & ~next(s6) :=1;}
```

Fig. 5. SMV code for SFC of Fig. 3

```
default next(token_overflow) := 0;
 in case{
     s1 & s4          :=1;
     s2 & s1          :=1;
     s3 & s1          :=1;
     s4 & s1          :=1;
     s5 & (s2 | s3)   :=1;
    ~s5 & (s2 & s3)   :=1;
     s6 & s3          :=1;
     s7 & (s5 & s6)   :=1;}
```

Fig. 6. SMV code for token overflow

```
(* SPEC token overflow *)
SPEC AG ! token_overflow

(* SPEC proper synchronization *)
SPEC EF (s5 & s6)
```

Fig. 7. SMV code for verification conditions

producing more tokens then allowed. If the sequential branches from s_2 and s_3 do not overlap anymore, the synchronization fails as well.

The counter examples delivered by the model checker show the evolution of the tokens from the initial state (one token in the initial step) up to the point of failure. This information does not show directly which parts of the structure make the SFC "unsafe" or "unreachable", but helps to trace the problem.

Since our model of Simple SFCs focuses on the structure

and abstracts away from data and actions, the complexity of the resulting SMV model is relatively small. The computation times of the model checker stay within a few seconds, even for large SFCs.

VII. Conclusions

This is the first work that presents an approach to determine safe SFCs. A characterization of this phenomenon, as well as a semantic definition is given. Moreover, an algorithmic solution was developed that allows to check for safe SFCs automatically.

There are several existing approaches to the verification of SFCs [12], [13], [14]. However, none of them is tailored to automatically determine safe SFCs. The advantage of our tailored approach in contrast to a general verification approach is that checking for safe SFCs does not require any user interaction at all. Hence, it can easily be embedded in analysis and design tools and serve as a back-end for automatic checks.

An optimal symbiosis of these two verification approaches might look like this: A PLC programming environment, which usually provides means for the design and programming of SFCs as well as simulation capabilities, is extended with both verification approaches. The checking for safe SFCs is started automatically before any compilation or simulation of an SFC, along with the usual consistency checks, e.g., type checking. Counter examples found during model checking can be visualized in the simulator, helping to find the structural error. Since the general verification approach requires the user to provide formal specifications which are to be checked, it is only started on request by a user with knowledge in formal methods.

Acknowledgment

This work was supported by the German Research Council within the DFG Priority Programme "Integration of Software Specification Techniques for Applications in Engineering" under grants RO 1122/10-2 and EN 152/32-2.

References

[1] D. Morley, "The history of the PLC," http://www.barn.org/FILES/historyofplc.html.

[2] R. Lewis, *Programming industrial control systems using IEC 1131-3*, revised ed., ser. Control Engineering Series. Stevenage, United Kingdom: The Institution of Electrical Engineers, 1998, vol. 50.

[3] F. Bonfatti, P. Monari, and U. Sampieri, *IEC 1131-3 Programming Methodology*, 1st ed. Fontaine, France: CJ International, 1999.

[4] *Programmable Controllers – Programming Languages, IEC 61131-3*, 2nd ed., International Electrotechnical Commission, Technical Committee No. 65, Nov. 1998, committee draft.

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

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

[7] N. Bauer and R. Huuck, "A parameterized semantics for sequential function charts," in *Workshop of Semantic Foundations of Engineering Design Languages (SFEDL)*, April 2002, satellite event of ETAPS 2002.

[8] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons for branching time temporal logic," in *Logics of Programs Workshop, IBM Watson Research Center, Yorktown Heights, New York, May 1981*, ser. LNCS, D. Kozen, Ed., vol. 131. Springer-Verlag, 1982, pp. 52–71.

[9] J.-P. Queille and J. Sifakis, "Specification and verification of concurrent systems in CESAR," in *Proceedings of the 5th International Symposium on Programming, Turin, April 6–8, 1982*, M. Dezani-Ciancaglini and U. Montanari, Eds. Springer-Verlag, 1982, pp. 337–350.

[10] R. Jhala and K. L. McMillan, "Microarchitecture verification by compositional model checking," in *Computer Aided Verification 13th International Conference, CAV 2001*, ser. LNCS, G. Berry and H. Comon, Eds., vol. 2102, 2001, pp. 396–410.

[11] S. A. Kripke, "Semantical considerations on modal logic," *Acta Philosophica Fennica*, vol. 16, pp. 83–94, 1963.

[12] D. L'Her, P. L. Parc, and L. Marcé, "Proving sequential function chart programs using automata," in *Proceedings of 2nd AMAST workshop on Real-Time Systems*, 1995.

[13] S. Lampérière-Couffin and J.-J. Lesage, "Formal verification of the sequential part of PLC programs," in *WODES 2000: 5th Workshop on Discrete Event Systems, Ghent, Belgium, August 21–23, 2000*, 2000.

[14] N. Bauer and R. Huuck, "Towards automatic verification of embedded control software," in *Asian Pacific Conference on Quality Software*, ser. IEEE, December 2001.