Verification of Sequential Function Charts Using SMV*

Sébastien Bornot · Ralf Huuck · Ben Lukoschus
Institute of Computer Science and Applied Mathematics
University of Kiel, Preußerstr. 1–9, D-24105 Kiel, Germany

Yassine Lakhnech
Verimag, Centre Equation, 2 av. de Vignate, F-38610 Gières, France

Abstract *Sequential function charts (SFCs) are defined as a modeling language in the IEC 1131-3 standard [1] and can be used to structure and drive programmable logic controllers (PLCs). It includes interesting concepts as hierarchy, history variables and priority. As the typical application area of this language is the control of industrial processes, it is obvious that safety and reliability is a crucial goal for systems using SFCs. In this work we give an abstract formal model for SFCs and present a method to automatically verify properties concerning the safety of such systems using the model checking tool SMV (Symbolic Model Verifier) [2, 3].*

Keywords: programmable logic controllers, sequential function charts, hierarchy, model checking, SMV

1 Introduction

The market for industrial process control and instrumentation has been continuously and rapidly growing within the last ten years and is expected to grow further in the future. Programmable logic controllers (PLCs) are the backbone of most automation projects and their market is expected to increase in even greater portion. Among the industries where they are used for controlling purposes are power generation, steel production, water, chemical and petrochemical [4].

Although PLCs are widely used, hardly any verification tools are available for them. This contribution aims at providing one solution for a high level validation of PLC programs and, thus, increasing the safety and reliability of PLC software. We focus on the structuring and programming language called *sequential function charts,* (SFCs), and present a model-checking [5, 6] approach. Therefore, we provide a translation from an abstract model of SFCs to the input language of the Symbolic Model Verifier (SMV) [2, 3]. This enables the validation of SFCs by verifying CTL formulae [7].

The remainder of this work is organized as follows: In order to give a clear idea of SFCs, we first introduce them informally in Section 2. Subsequently, we present the formal syntax in Section 3 which later will be translated to the SMV syntax and briefly explain their semantics in Section 4. The translation itself as well as an illustrating example is described in Section 5. Finally, we draw some conclusions and discuss related work.

2 The SFC Language

SFCs are defined in [1] as elements of a graphical programming and structuring language for PLCs. The SFC elements provide means of partitioning a PLC program organization unit into *steps* and *transitions.* Each SFC has exactly one *initial step.* Associated which each step is a sequence of *actions,* which are either instructions in one of the other programming languages proposed by the standard or again an SFC. Hence, it is possible to have a nesting of SFCs. Moreover, with each transition a *transition condition*, also called *guard,* is associated. An example of an SFC is depicted in Fig. 1.

*This work has been partially supported by the European Community in the Esprit Long-Term Research Project 23498 "VIRES" (Verifying Industrially Relevant Systems).

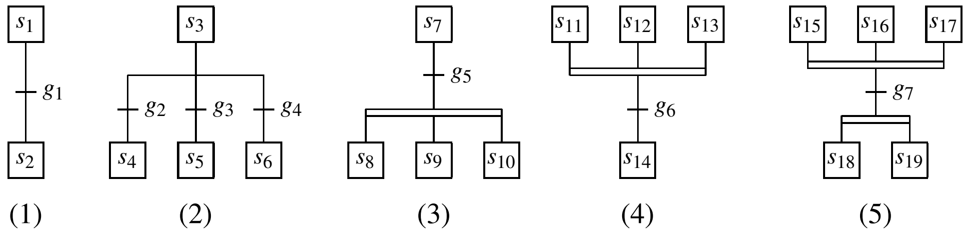

Figure 2: Basic transition types.

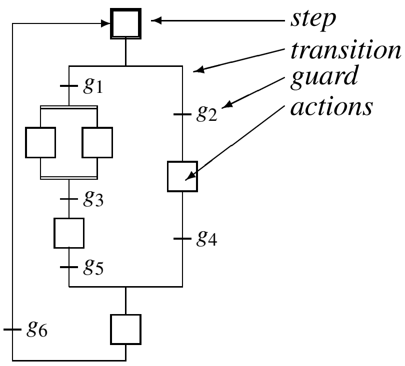

Figure 1: Elements of sequential function charts.

For SFCs we can identify a number of different transition types (cf. Fig. 2). Transition (1) denotes a *simple transition* between two steps, (2) describes *alternative branching*, i.e., the choice between several transitions, (3) *divergence*, (4) *convergence*, and (5) a simultaneity of both of the former. In this context divergence means a parallel branching from one step to a set of next steps while convergence means the synchronization of several steps in parallel to a single one. However, direct combinations of both as depicted in (5) are allowed. Moreover, the standard refers to transitions like the one guarded by g_6 in Fig. 1 as *loops*. From our point of view loops as well as alternative branching belong to the same class, namely (sets of) simple transitions. These basic transition types can be combined into more complex structures; see Fig. 1 and 3 for some examples.

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

In the following we describe syntax and semantics for SFCs as basis for the formal verification.

3 Abstract Syntax of SFCs

PLCs can be seen as technical devices reacting on certain inputs and their internal state with some specific output. Sequential function charts are programs running inside the PLCs. In every PLC cycle the input information is written to some input variables. Then some computation is performed based upon these and internal variables. Finally, the output is written to some output variables. To describe the values of all variables we use the notion of a *state,* which is a function assigning a value to each variable.

Definition 1 (State) Let Σ be a nonempty set of elements σ, called *states.* □

A state (i.e., the values of the variables) can be changed by *state transformations.* In [1] different types of programming languages are defined for such state transformations. We abstract from this by saying that a state transformation is a function that transforms a given state into another state.

Definition 2 (State transformation) A function $f : \Sigma \to \Sigma$ is called *state transformation*. Let F be the set of all state transformations. □

Each step of an SFCs is labeled with a (possibly empty) sequence consisting of *actions,* which are either state transformations or calls of nested SFCs.

Definition 3 (Action, action sequence) An *action* is either a state transformation $f \in F$ or an SFC $\mathcal{S}$ (defined below). We call a finite sequence $\langle a_1, \ldots, a_k\rangle$ of actions (including the empty sequence with $k = 0$) an *action sequence.* Let A be the set of all action sequences. □

We define a *guard* as a set of states, which can be seen as a Boolean expression describing a set of possible values for each variable.

Definition 4 (Guard) A *guard* g is a subset of Σ. Let G be the set of all guards. Given $g \in G$ and a state $\sigma \in \Sigma$ with $\sigma \in g$, we say that *"σ satisfies g"* or *"g is valid in σ"*, denoted by $\sigma \models g$. □

Next, we present the formal syntax for SFCs. Key points are orderings on the steps as well as on the transitions. These allow to determine in which order actions are executed and which transitions will be taken if several are in conflict with each other. Let *SFC* be the set of sequential function charts, which are defined next.

Definition 5 (SFC) A *sequential function chart* (SFC) is a 6-tuple $\mathcal{S} = (S, s_0, T, a, \sqsubset, \prec)$, where S is a finite set of *steps,* $s_0 \in S$ is the *initial step,* $T \subseteq (2^S \setminus \{\emptyset\}) \times G \times (2^S \setminus \{\emptyset\})$ is a finite set of *transitions,* $a : S \to A$ is an *action labeling function* which assigns an action sequence to each step, $\sqsubset \subseteq S \times S$ is an irreflexive total order on steps, used to define the order in which the action sequences of the active steps are to be executed, and $\prec \subseteq T \times T$ is an irreflexive partial order on transitions, such that conflicting transitions are comparable. This defines priorities for solving the conflicts. □

The transitions in T can be divided into the basic types as introduced and depicted in Fig. 2. Formally, we uniquely represent a transition by its set of source steps, its guard and its set of target steps. Referring to Fig. 2 number (1) will be represented by $(\{s_1\}, g_1, \{s_2\})$ and (4) by $(\{s_{11}, s_{12}, s_{13}\}, g_6, \{s_{14}\})$. Note that (2) has a representation by three simple transitions.

When putting a graphically given SFC into formal syntax the main task is to determine the orders $\sqsubset$ and $\prec$. From the standard and practical investigations we can derive the following properties which have to be satisfied for them: The transitions of alternative branches are ordered from "left to right", i.e. the left branch has the highest priority, if not otherwise stated by explicit priorities. The steps are also ordered from left to right, this holds especially for parallel branches.

4 Informal Semantics of SFCs

This section provides an informal description of the semantics for the SFCs defined in the previous section. A formal semantics is defined in [8].

Let $\mathcal{S}$ be an SFC. We describe its semantics by explaining how the *configuration* of $\mathcal{S}$ changes during execution. A configuration consists of a state σ (describing the current variable values) and a set *active* containing all the steps which are active.

In the initial configuration all variables have some default value, and the only steps that are active are the initial steps of the SFCs. Then the following sequence, called *cycle,* is executed repeatedly for the top-level SFC $\mathcal{S}$:

1. Get new input from the environment and store it into the input variables, i.e., in some designated part of the current state σ.
2. Execute the sequence of actions in the active steps, modifying the variable values in σ. If there is more than one step active, sort the active steps using the ordering $\sqsubset$ and execute the action sequences in that order. If another SFC is nested inside an active step, i.e., one of the actions is an $\mathcal{S}' \in SFC$, perform items 2 and 3 of the cycle for the nested SFC $\mathcal{S}'$.
3. Determine the transitions that will be taken by evaluating the guards of the transitions using the new state σ (after execution of the actions in item 2), deactivate the source steps of these transitions (remove them from *active*), and activate the destination steps (add them to *active*). If some enabled transitions in alternative branches are conflicting, i.e., have some source step(s) in common, only take the transition with the lowest position in the $\prec$ ordering.
4. Send the values of the output variables in the new state σ to the environment.

An additional language feature of SFCs allows guards of transitions to reason about the activity of steps, i.e., guards can be enabled and disabled by other steps becoming active or inactive. In our framework, we add internal Boolean variables to the state, called *step variables.* These variables cannot be modified by the actions, but implicitly change whenever a transition is taken and steps are exited or entered.

5 Verification

When analyzing SFCs, one is very often interested in the verification of reachability properties. This means, e.g., to check an SFC for mutual exclusion like "steps s_1 and s_2 are never active simultaneously" or for causalities like "whenever an input x occurs, step s_3 will eventually become active".

In this section we present a way to do so automatically for given properties. Therefore, we first translate SFCs into the input language of the model checking tool SMV [2, 3] and formulate verification properties in the temporal logic CTL [7]. Then, the verification is performed by SMV.

We have the following assumptions: Since we focus on reachability of steps, we are not interested in output variables, and also abstract from all internal variables except for the Booleans referring to step activities. Hence, in the following we consider SFCs with the following structure:

- Each step contains no state transformations and at most one nested SFC, i.e., $a(s)$ is either the empty sequence or a sequence which contains one single SFC (which can have nesting itself).
- Guards only describe input variables or the activity of steps.

Any given SFC can be transformed to fulfill these restrictions such that the resulting SFC is an abstraction of the original one.

5.1 Running Example

Fig. 3 shows a top-level SFC with 5 steps. In each of the steps s_2 and s_3 we have a nested SFC, shown on the right side. Both nested SFCs have the same structure. The guard $sub_3.s_8.X$ of the transition from s_2 to s_4 in the top-level SFC is enabled if and only if step s_8 of the SFC nested in s_3 is active.

5.2 Preprocessing

In an alternative branching (cf. (2) in Fig. 2) conflicts occur if some of its transitions are simultaneously enabled. These conflicts are solved by the ordering $\prec$ we imposed on the set T of transitions. To simplify the translation to SMV, we modify the guards of the transitions such that there are no more

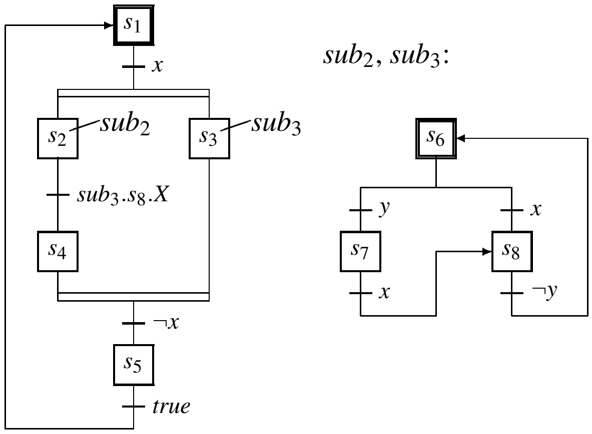

Figure 3: Running example.

conflicts. We enable a transition only if its guard holds and no other higher-priority transition which shares at least one common source step (these are the conflicting transitions) is enabled.

This is done by simultaneously replacing in each transition $t = (Q, g, Q') \in T$ its guard g by

$$g' = g \bigwedge_{\substack{t_1=(Q_1,g_1,Q'_1)\in T,\\ Q_1\cap Q\neq\emptyset,\ t_1\prec t}} \neg enab(t_1), \qquad (1)$$

where $enab(t_1)$ expresses that t_1 is enabled, i.e., its source steps are active and its guard evaluates to *true*. This replacement step produces an equivalent SFC which does no longer need the ordering $\prec$, since all transitions of alternative branchings are now disjoint, and thus, no conflicts can occur.

In Fig. 3 we only have alternative branching in the transitions leaving s_6 of each nested SFC. The guard of the transition to s_7 remains unchanged, since it has the higher priority (default ordering "from left to right"), and thus, the conjunction in (1) is empty. The guard of the right transition to s_8 is changed to $x \wedge \neg(s_6.X \wedge y)$, where $s_6.X \wedge y$ describes the enabledness of the left transition.

5.3 Translation to SMV

Let $\mathcal{S} = (S, s_0, T, a, \sqsubset, \prec)$ be an SFC with the restrictions above and with the preprocessing step already performed. The translation of $\mathcal{S}$ to SMV is based on a few key ideas. SMV code represents a transition system whose states are given as evaluations of a set of variables. In our case, we will use these transitions to represent a complete cycle of the SFC. Thus, we introduce a variable for each input variable, a Boolean variable for each step and a module for each (nested) SFC. These variables represent the configuration of the SFC at the beginning of a cycle, that is, the step activity *before* any transitions are taken and the *old* values of the input variables (which were read in the last cycle). The definition of SMV transitions also allows us to reason about these values in the beginning of the next cycle (or end of the current cycle, which is the same) by using the `next` operator.

In the following we describe in detail the translation with reference to the running example.

5.3.1 Global Definitions

Any SMV input file contains a `main` module. Each input variable of $\mathcal{S}$ becomes a variable of the `main` module, in our example x and y. References to steps outside the scope of $\mathcal{S}$ are also considered as (Boolean) input variables.

The top-level SFC $\mathcal{S}$ is defined later as the module `sfc0`. It is instantiated as the variable `sfc_top` of the `main` module. The first two parameters restrict the activity of the SFC, which is explained in detail below. Here both are 1 (meaning *true*), since the top-level SFC is always active. The remaining parameters list all the variables that may be read by the module and its sub-modules. For the top-level SFC these are only the input variables.

```
MODULE main
  VAR  x : boolean;  y : boolean;
  VAR  sfc_top : sfc0(1,1,x,y);
```

5.3.2 Translation of the SFCs

In this section we translate each SFC (top-level and the nested ones) into an SMV module. The module for the top-level SFC is denoted by `sfc0` and the nested ones will be called `sfc`n, each with a unique $n > 0$. For each SFC $\mathcal{S}'$ (top-level or nested), we put the following into the SMV file (the SMV code shown is the module for the top-level SFC of our running example).

The module header has two parameters controlling the activity of the SFC and a list of variables that may be read by the module and its sub-modules:

```
MODULE  sfc0(parent_act,step_act,x,y)
```

The first parameter indicates whether the parent SFC is active, and the second one if the higher-level step it is nested in is active. If one of these two Booleans is *false*, no transitions can be taken in this SFC. The rest of the parameter list contains all the variables that may be read by the module and its sub-modules. These are the input variables and step variables of higher-level SFCs we want to refer to in guards. This list is necessary because SMV only allows to read local module variables and variables of nested modules, but no higher-level variables.

Next we define the macro `active`, which combines the two parameters mentioned above and thus states that transitions in this SFC can be taken. When translating the transitions to SMV, we add `active` to every guard, so no transitions will be enabled if this SFC becomes inactive.

```
DEFINE  active := parent_act & step_act;
```

For each step in $\mathcal{S}'$, we define a Boolean step variable denoting if the step is active:

```
VAR  s1 : boolean;  s2 : boolean;
     s3 : boolean;  s4 : boolean;
     s5 : boolean;
```

For each SFC that is nested in a step of $\mathcal{S}'$ the corresponding module is instantiated and the activity of $\mathcal{S}'$ as well as the step variable, i.e., the step it is nested in, are provided as parameters. Furthermore, we hand over the variables mentioned above. In our running example we have:

```
VAR  sfc_top_s2 : sfc1(active,s2,x,y);
     sfc_top_s3 : sfc1(active,s3,x,y);
```

Moreover, the step variables introduced above have to be initialized. For the initial step of $\mathcal{S}'$ its variable is set to true and for all other to false:

```
ASSIGN  init(s1) := 1;  init(s2) := 0;
        init(s3) := 0;  init(s4) := 0;
        init(s5) := 0;
```

Finally, the transitions are defined, i.e., how the step variables describing the activities evolve. A step s is active in the subsequent configuration if s will be entered (i.e., there exists a transition t such that s is a target step of t and t is enabled, i.e., all its source steps are currently active and the guard of this particular transition holds) or if s is currently active and no outgoing transition is enabled.

Note that the "or" is not exclusive and that no transition is enabled if the parent SFC of $\mathcal{S}'$ is not active or its step containing $\mathcal{S}'$ is not active (we add `active` to all guards to ensure this). Fig. 4 shows the resulting SMV code for the top-level SFC.

We use the `next` values of step variables whenever we reason about steps of an SFC whose transitions are taken before the transition we are just defining. This can be decided during translation by looking at the nesting structure and the step orderings $\sqsubset$ in the SFCs. In our example, transitions in the left nested SFC are taken first, then transitions of the right nested SFC and then the transitions of the top-level SFC. If the nested SFCs would use each other's step variables in some of its guards, this would lead to different `ASSIGN` declarations for each nested SFC. In this case, we have to define one exclusive module for each of these SFCs.

For any access to input variables we also use the `next` value, since we are interested in the new input that will be read during the current cycle.

5.4 Checking Reachability Properties

In this section SMV is used to check reachability properties written as CTL formulae. In order to do so, we add the specifications to the SMV input file produced in the previous section using `SPEC` declarations and run SMV on the file. In the following we list some classes of properties and illustrate the verification of such properties by our example.

Simple reachability. In order to check that a step s_i is reachable, we add the specification

```
SPEC  EF si
```

to the input file. The formula states that there exists a finite computation prefix which reaches a configuration where s_i is active. SMV shows that this is true for all steps of all three SFCs.

Repeated reachability (lifeness, no deadlock). A sensible property of any SFC is that it does not deadlock, i.e., never reaches a configuration from where no transitions can be taken, no matter how the environment behaves. Hence, for every step s_i whenever it is active there exists a computation which leads to a state where s_i is inactive again:

```
SPEC  AG (si -> EF !si)
```

Again, SMV states that this is true for all steps.

Combined reachability (synchronization, mutual exclusion). Another class of properties can be used to check if several steps are active at the same time. Let us assume that the actions (which we abstracted from in our approach) of step s_4 of the top-level SFC and step s_8 of the right nested SFC have to synchronize repeatedly, i.e., they have to take place in the same cycle. This is expressed by the fact that there is always a configuration reachable where both steps are active.

```
SPEC  AG EF (s4 & sfc_top_s3.s8)
```

This is proven to be valid by SMV. Next, we check if these steps are *always* active at the same time:

```
SPEC  AG (s4 <-> sfc_top_s3.s8)
```

This specification does not hold, and SMV provides a counterexample showing that there is a configuration reachable where s_8 is active but s_4 is not.

Finally, we check if the actions in step s_4 of the top-level SFC and in step s_8 of the right nested SFC should be allowed to access a common resource which has to be used exclusively by at most one of both SFCs. This mutual exclusion property

```
SPEC  AG !(s4 & sfc_top_s3.s8)
```

does of course not hold, since it contradicts the repeated synchronization above.

Causal dependencies with input variables. We also can examine causal dependencies between input variables and reachability. E.g., in order to show that the priority for the alternative branching in the nested SFCs is working correctly, we use

```
SPEC  AG ((active & s6) ->
          (AX (x & y -> s7 & !s8)))
```

to prove that the nested SFC and s_6 being active implies that at the beginning of the next cycle $x = y = true$ implies the transition to s_7 has been taken and not the one to s_8. This is valid for both

```
ASSIGN
  next(s1) := (s5 & active & 1) | (s1 & !(s1 & active & next(x)));
  next(s2) := (s1 & active & next(x)) | (s2 & !(s2 & active & next(sfc_top_s3.s8)));
  next(s3) := (s1 & active & next(x)) | (s3 & !(s3 & s4 & active & !next(x)));
  next(s4) := (s2 & active & next(sfc_top_s3.s8)) | (s4 & !(s3 & s4 & active & !next(x)));
  next(s5) := (s3 & s4 & active & !next(x)) | (s5 & !(s5 & active & 1));
```

Figure 4: SMV transitions for the top-level SFC.

nested SFCs. Note that if `active` is removed from the specification above, the formula is no longer valid, since if we get the input $x = y = \mathit{true}$ in the very first cycle, the transitions from s_6 to s_7 cannot be taken, since the nested SFCs are not active yet.

6 Conclusions and Related Work

We have presented a method to make symbolic model checking accessible for SFCs. By an effective translation of the SFC syntax into SMV source code we gained the whole richness of CTL model checking. Moreover, we kept the structure of the original SFC as well as the possibility to reason on step variables. In comparison to related work [9, 10] we incorporated both the concept of hierarchy and the concept of priorities on transitions.

A future research topic is to apply this method on exhaustive case studies and improve the efficiency of model checking SFCs by exploiting their structural properties like hierarchy and reasoning on step variables. Moreover, an extension to a more sophisticated real-time model is possible.

References

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

[2] K.L. McMillan. *The SMV system*. Carnegie-Mellon University, February 1992. Draft version describing SMV revision 2.2.

[3] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[4] R.W. Lewis. *Programming industrial control systems using IEC 1131-3*, volume 50 of *Control Engineering Series*. The Institution of Electrical Engineers, revised edition, 1998.

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

[6] E.M. Clarke, E.A. Emerson, and E. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach. In *10th ACM symposium of Programming Languages*. ACM Press, 1983.

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

[8] S. Bornot, R. Huuck, Y. Lakhnech, and B. Lukoschus. An abstract model for sequential function charts. In *WODES 2000: 5th Workshop on Discrete Event Systems, Ghent, Belgium, August 21–23, 2000*, 2000.

[9] 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.

[10] D. L'Her, P. Le Parc, and L. Marcé. Modeling and proving grafcets with transition systems. In *2nd AMAST workshop on Real-Time Systems and Project 'Models and Proofs', Bordeaux, France, June 14–16, 1995*, 1995.