AN ABSTRACT MODEL FOR SEQUENTIAL FUNCTION CHARTS

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

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

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

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

Keywords: programmable logic controllers, sequential function charts, hierarchy, semantics

Abstract Sequential function charts (SFCs for short) is a modeling language defined in the IEC 1131-3 standard that can be used to drive programmable logic controllers (PLCs). It includes interesting concepts like hierarchy and history variables. As the typical application area of this language is the control of industrial processes, it is obvious that a clear understanding of the semantics of these languages is of interest. It is also of interest for the programmers as well as for the builders of development environments based on this standard to know which semantic ambiguities the standard leaves open. This paper addresses these points providing a clear and formal semantic framework for SFCs that can be used, for instance, as a sound basis for building validation tools.

Introduction

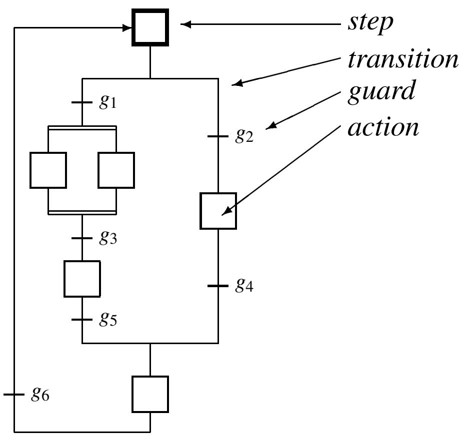
The market for industrial process control and instrumentation has been continuously and rapidly growing within the last ten years and is expected to hold on for the future. Programmable logic controllers (PLCs) are the backbone of most automation projects and their market is expected to increase in even greater portion. Among the industries where they are used for controlling purposes are power generation, steel production, water, chemical and petrochemical [Lewis, 1998]. Moreover, in the past many vendors for PLC systems used their own programming languages which were incompatible with others. For re-using components, maintenance, and inter-operateability, a clear notion for each language and its semantics is important. By 1993 the IEC 1131-3 [IEC, 1998] standard was introduced to unify and compromise the main different approaches. However, no formal semantics for the different program languages were given and, as we will point out, there are ambiguities which might be crucial for the safety of a system.

In this paper we give an abstract formal semantics for the programming and structuring language called sequential function charts (SFCs) in order to provide a clear understanding of this language. Moreover, this formal framework will serve as a basis for future formal reasoning. The remainder of this paper is structured as follows: First, we will introduce sequential function charts informally in Section 1. In Section 2 we give several examples to point out semantical ambiguities. In order to solve cases like these, we introduce a formal syntax and semantics in Section 3 and 4 and apply them in Section 5 to the previous examples. Finally, we draw some conclusions and point out some directions for future work.

1. THE SFC LANGUAGE

Sequential function charts are defined in [IEC, 1998] as elements of a graphical programming and structuring language for programmable logic controllers. The SFC elements provide a means of partitioning a programmable controller program organization unit into *steps* and *transitions.* For every SFC there exists 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 sequential function charts. Moreover, which each transition a *transition condition*, also called *guard,* is associated. An example of a sequential function chart is depicted in Figure 1.

For SFCs we can identify a number of different transition types (cf. Figure 3). 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



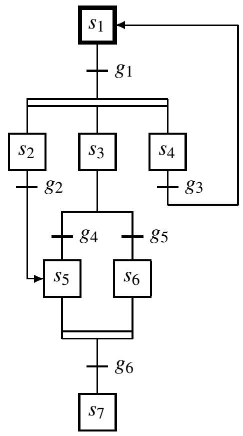


Figure 1 Elements of SFCs.

Figure 2 An ill-structured SFC.

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 transition like the one guarded by g_6 in Figure 1 as *loops*. From our point of view loops as well as alternative branching belong to the same class, namely (sets of) simple transitions.

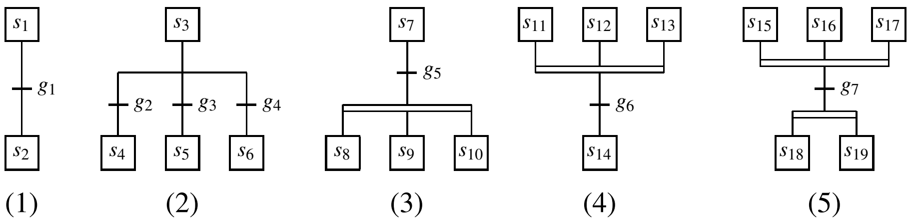


Figure 3 Basic transition types.

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 the transitions are ordered "from left to right" in decreasing priority.

These basic transition types can be combined into more complex transition structures like in Figure 1 and 2. However, there are various combinations which do not seem to make sense, e.g., in the SFC shown in Figure 2. There we have the transition $(\{s_4\}, g_3, \{s_1\})$ which jumps out of a simultaneous sequence, the transition $(\{s_2\}, g_2, \{s_5\})$, which jumps from one simultaneous branch to another, and the transition $(\{s_5, s_6\}, g_6, \{s_7\})$, which is a convergence of simultaneous sequences of two steps which are part of a sequence selection starting in step s_9.

Although the standard forbids to use such kinds of "unsafe" or "unreachable" SFCs, it does not give a precise characterization of this phenomenon. For the time being we do not want to pose any explicit restrictions on the SFC syntax. Mainly, because we want to allow as much freedom as possible and because there are commercial tools available which "support" SFCs with these kinds of transitions and do simulations on them. Hence, in the following we define a semantics for sequential function charts which can also cope with such constructions.

2. AMBIGUITIES IN THE SEMANTICS

The standard defines some rules how to build an SFC from the aforementioned basic elements and defines some evaluation rules. But against common belief the semantics for such an SFC is sometimes far from obvious. In order to illustrate this claim, we will present some examples. First of all, let us take a look at Figure 4.

From the initial step (the double framed box) control will only advance if the guard $x = 0$ is satisfied. Next, control will simultaneously reside in the locations with associated actions $x := 1$ and $x := 2x$. Now, there are the following options how to proceed:

1 The left action $x := 1$ will be performed and the left transition guarded by $x \leq 1$ taken, afterwards the action on the right will be performed. Depending on whether the second action uses already the new value for x (which is one) or the old one (which is zero) the right transition guarded by $x = 2$ will be taken or not.

2 The right action will be performed first, the right guard still evaluates to *false*, but then the left action will be performed and again, the left transition will be taken.

3 The left action will be performed first, then the right one and afterwards the guards will be checked. Hence, the right transition will be taken.

4 The execution order is vice versa and hence, the left transition will be taken.

Let us take a look at another ambiguous SFC at Figure 5. Here it is not clear what will happen if control reaches the step with a nested SFC. Will the nested one be executed until it cannot proceed anymore? Has the top level one higher priority? How does the nested one affect variables of the top level one? Which

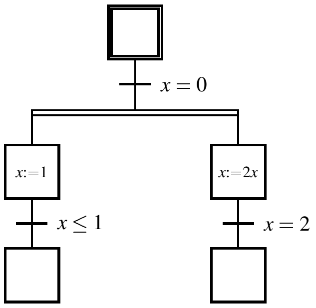


Figure 4 Order of Execution?

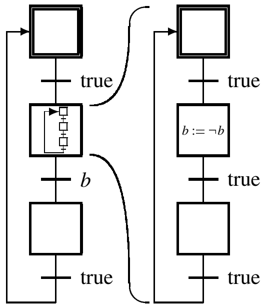


Figure 5 How to deal with hierarchy?

step of the nested SFC will be entered if we reach it for the second or third time? Again, there is more than one option.

Thus, there are quite a few possible semantics. However, the standard is not clear about it as well as it is not for a number of other cases. Therefore, it is necessary to point to these ambiguities and to give a formal semantics which provides a unique solution.

3. ABSTRACT FORMAL SYNTAX

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. Afterwards, 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 [IEC, 1998] different types of programming languages are defined for such state transformations. We abstract from these types by simply saying that a state transformation is a function that transforms a given state into another state. Here state transformations are deterministic; in nondeterministic languages a state is transformed into one of several possible successor states.

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

Each step of an SFC 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 or an SFC $\mathcal{S}$ (defined below). We call a finite sequence $\langle a_1, \dots, a_k \rangle$ 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. In a syntactical setup which uses variables, a guard is 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 a guard $g \in G$ 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 sequential function charts. 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 is used to define the priority for conflicting transitions.*

The transitions in T can be divided into the basic types as introduced and depicted in Figure 3. Formally, we uniquely represent a transition by its set of source steps, its guard and its set of target steps. Referring to Figure 3 number (1) will be represented by $(\{s_1\}, g_1, \{s_2\})$ and (4) by $(\{s_{11}, s_{12}, s_{13}\}, g_6, \{s_{14}\})$. Remember that (2) has a representation by three simple transitions.

When putting a sequential function chart 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. ABSTRACT FORMAL SEMANTICS

In this section we provide a semantics for the SFCs defined in the previous section. Let $\mathcal{S} = (S, s_0, T, a, \sqsubset, \prec)$ be an SFC, and let $\mathcal{S}_i = (S_i, s_{0,i}, T_i, a_i, \sqsubset_i, \prec_i)$,

$i = 1,\ldots,n$, be the SFCs nested recursively inside the steps of $\mathcal{S}$. For a global, flat access to the nested structure we define $\bar{S} = S \cup S_1 \cup \ldots \cup S_n$.

There are two different things we have to keep track of when observing executions of $\mathcal{S}$. First, we need information about the current state of $\mathcal{S}$, i.e., the values of its variables. And we must know in which steps of $\mathcal{S}$ and its sub-SFCs $\mathcal{S}_1,\ldots,\mathcal{S}_n$ control resides.

It is crucial to notice that there is a difference between control residing in a step and steps whose actions are actually performed. The former we will call *ready* steps, and the latter *active* steps. Each active step is also a ready step, but the converse does not hold, since actions in ready steps of nested SFCs will only be performed if the step of the top-level SFC containing the nested SFC is active. We can always deduce the active steps from the set of ready steps.

We store the information about the state and the ready steps of $\mathcal{S}$ in a *configuration:*

Definition 6 (Configuration) *A* configuration *of $\mathcal{S}$ is a pair $(ready,\sigma)$, where $\sigma \in \Sigma$ is a state and $ready \subseteq \bar{S}$ is the set of* ready *steps.*

Such a configuration is changed in the *cycles* of an SFC. A cycle of an SFC performs the following sequence:

1 Get new input from the environment and store the information into the state σ.

2 Execute the action sequences in the top-level ready steps, changing the state σ. If another SFC is nested inside a ready step, i.e., one of the actions in an action sequence is an $\mathcal{S}' \in \mathit{SFC}$, all ready steps of $\mathcal{S}'$ are also active, and we perform items 2 and 3 for the nested SFC $\mathcal{S}'$.

3 Determine the set of next ready steps by looking at the current top-level ready steps and evaluating the guards of the transitions, and then taking the transitions by changing the set *ready*.

4 Send the outputs to the environment by extracting the required information from the new state σ.

We do not specify formally how to interact with the environment; we only consider items 2 and 3.

We define the semantics for SFCs by showing how a configuration changes when one SFC cycle is executed. That is, given a configuration $(ready,\sigma)$, with σ already containing the new input information from the environment, we compute the next configuration $(ready',\sigma')$ by executing the action sequences inside the ready steps which are active and by computing the new set of ready steps. Then the output information for the environment can be extracted from σ'. This single-cycle semantics, which is defined next, can easily be extended

$$
\frac{\begin{array}{l}
\{s_1,\ldots,s_m\} = S \cap \mathit{ready} \\
s_1 \sqsubset s_2 \sqsubset \ldots \sqsubset s_m \\
\mathcal{F}(\langle a_1,\ldots,a_k\rangle)(\mathit{ready},\sigma) = (\mathcal{F}'(a_k)\circ\mathcal{F}'(a_{k-1})\circ\ldots\circ\mathcal{F}'(a_1))(\mathit{ready},\sigma) \\
\mathcal{F}'(a_i)(\mathit{ready},\sigma) = \begin{cases} (\mathit{ready}, a_i(\sigma)), \text{ if } a_i \in F \\ (\mathit{ready}',\sigma') \quad, \text{ if } a_i \in \mathit{SFC} \wedge a_i :: (\mathit{ready},\sigma) \longrightarrow (\mathit{ready}',\sigma') \end{cases} \\
(\mathit{ready}_1,\sigma_1) = (\mathcal{F}(a(s_m))\circ\mathcal{F}(a(s_{m-1}))\circ\ldots\circ\mathcal{F}(a(s_1)))(\mathit{ready},\sigma) \\
\mathit{enabled} = \{(Q,g,Q') \in T \mid Q \subseteq \mathit{ready}_1 \wedge \sigma_1 \models g\} \\
\mathit{taken} = \{t = (Q,g,Q') \in \mathit{enabled} \mid \neg\exists t_1 = (Q_1,g_1,Q_1') \in \mathit{enabled} : Q \cap Q_1 \neq \emptyset \wedge t_1 \prec t\} \\
\mathit{ready}' = (\mathit{ready}_1 \setminus \bigcup_{(Q,g,Q')\in \mathit{taken}} Q) \cup \bigcup_{(Q,g,Q')\in \mathit{taken}} Q' \\
\sigma' = \sigma_1
\end{array}}{\mathcal{S} :: (\mathit{ready},\sigma) \longrightarrow (\mathit{ready}',\sigma')}
$$

Figure 6 SFC single-cycle semantics rule.

to a multi-cycle semantics which computes a sequence of outputs from a given sequence of inputs from the environment.

Single-cycle semantics. We define a relation $\longrightarrow$ which describes configurations before and after performing items 2 and 3 above. The term $\mathcal{S} :: (\mathit{ready},\sigma) \longrightarrow (\mathit{ready}',\sigma')$ states that one cycle of the SFC $\mathcal{S}$ changes the configuration (ready,σ) to $(\mathit{ready}',\sigma')$.

Given the SFC $\mathcal{S} = (S, s_0, T, a, \sqsubset, \prec)$ with a configuration (ready,σ), the rule given in Figure 6 defines the configuration $(\mathit{ready}',\sigma')$ resulting from executing the action sequences inside ready steps in $\mathcal{S}$ (and maybe recursively executing action sequences and taking transitions in the sub-SFCs of $\mathcal{S}$) and then taking transitions of the top-level SFC $\mathcal{S}$.

The rules works as follows: First we compute the set of ready steps on the top level of $\mathcal{S}$ and put them in the order defined by $\sqsubset$, resulting in the sequence $s_1 \sqsubset s_2 \sqsubset \ldots \sqsubset s_m$. Then the action sequences $a(s_1),\ldots,a(s_m)$ inside these steps are executed in this order. To do so, we define the configuration transformer $\mathcal{F}$ which takes a sequence of actions $\langle a_1,\ldots,a_k\rangle$ and a configuration (ready,σ) and computes the configuration after executing these actions in the order $a_1,\ldots,a_k$. A single action is executed by a second configuration transformer $\mathcal{F}'$ which takes a single action a_i and a configuration to produce the resulting configuration. In case this action a_i is an element of *SFC*, i.e., there is another SFC nested inside the step we are looking at, $\mathcal{F}'$ recursively calls the rule to compute the new configuration. Note that in the latter case σ and *ready* may both be changed, whereas in the other case ($a_i \in F$) the set *ready* remains unchanged, since transitions in the top-level SFC are taken later.

These transformations on the configuration are done in the order of the steps given by $\sqsubset$. The result is stored in $(ready_1, \sigma_1)$.

Then transitions on the top-level SFC $\mathcal{S}$ are taken, resulting in a change of *ready*: A transition $(Q, g, Q') \in T$ is called *enabled* at the configuration $(ready_1, \sigma_1)$ if and only if all predecessor steps in Q are in $ready_1$ and the guard g of the transition is valid under σ_1. We store these transitions in the set *enabled*. But not every enabled transition is actually taken, since there may be conflicts. Two enabled transitions are called *conflicting,* if they share at least one common source step. In this case, the smallest transition w.r.t. the order $\prec$ (i.e., the transition with the highest priority) will be taken. So we define the set *taken* as the set of all transitions in *enabled* for which no other enabled transition with a higher priority (i.e., a lower position in $\prec$) exists.

Finally, the set of ready steps is updated by removing all source steps of the transitions in *taken* and then adding all the destination steps of the transitions in *taken*. Since the state does not change when taking transitions, we have $\sigma' = \sigma_1$.

5. SOLVING THE EXAMPLES

The previous section defined the semantics for sequential function charts formally. We use it in this section to solve the ambiguities from the introducing Section 2.

For the example in Figure 4 it is crucial to know that after computing the set of ready steps and putting them in order these steps are executed in this order. Since the order is given by "left to right" convention, we have in our case where control resides at the locations with the associated actions $x := 1$ and $x := 2x$, that the step labeled with $x := 1$ is preceding the step labeled with $x := 2x$ in the order $\sqsubset$. This means, the evaluation will be as follows in this situation: First, we compute $x := 1$. Next, we use the resulting state to compute $x := 2x$, which means we have the state where $x = 2$. Afterwards, the guards are evaluated and the set of ready steps are determined which are the steps $x := 1$ and the bottom step at the right.

The second example deals with the nested SFC depicted in Figure 5. First let us assume control resides at some step within the nested sequential function chart and the guard b of the top-level SFC does not hold. Then, in every cycle one step of the nested SFC is executed as given by the rule for single-cycle semantics. However, it becomes interesting if control resides in step $b := \neg b$, i.e., will toggle b to become *true*. First of all, the actions of the active steps are executed, i.e., b is toggled. Next, the corresponding guard of the nested step is evaluated. Since it is *true* the succeeding step will be added to the set of *ready* steps while the current one is removed. Going back in the recursive application of the semantics rule will yield in evaluating the top-level guard b. Since this one also holds its succeeding step is also added to *ready* while the current on comprehending the nested SFC is removed from it. This means, in the next cycle only the top-level sequential function chart is active and the nested one is suspended. But the next time control enters the nested one it will start from the ready set. This means the semantics keeps track of the history for each SFC.

The resolutions of semantical ambiguities presented here coincide with actual implementations in industrial development tools for PLC programming such as CoDeSys (by 3S-Smart Software Solutions GmbH, Germany) and ISaGRAF (by CJ International, France, enclosed in [Bonfatti et al., 1999]).

6. CONCLUSIONS

We have presented a formal abstract syntax and semantics for sequential function charts. Moreover, we illustrated by several examples that this formal framework is indeed necessary to avoid ambiguities not solved in the standard. The introduction of orders on steps and transitions enables us to parameterize the semantics, i.e., the behavior of the sequential function charts. Hence, our framework can easily adjust to different interpretations of the standard.

A future research topic is to incorporate action qualifiers as well as time into our framework. Moreover, we have developed a method to algorithmically classify SFCs which are syntactically "safe", i.e., are free of unsensible structures.

Acknowledgments

This work has been partially supported by the European Community in the Esprit Long-Term Research Project 26270 – VHS (Verification of Hybrid Systems). Furthermore, we thank Stefan Kowalewski for fruitful discussions.

References

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

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

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