A STOPWATCH SEMANTICS FOR HYBRID CONTROLLERS

Nanette Bauer [*,1] **Ralf Huuck** [**,1,2] **Ben Lukoschus** [**,1,3]

[*] *Department of Chemical Engineering, University of Dortmund, Germany.* `n.bauer@chemietechnik.uni-dortmund.de`
[**] *Institute of Computer Science and Applied Mathematics, University of Kiel, Germany.* {`rhu`,`bls`}`@informatik.uni-kiel.de`

Abstract: Programmable Logic Controllers (PLC) are frequently used in the automation industry for the control of hybrid systems. Although the programming languages for PLCs are given in the standard IEC 61131-3, their semantics are defined in an ambiguous and incomplete way. This holds in particular for the graphical language Sequential Function Charts (SFC), a high-level programming language comprising such interesting features as parallelism, activity manipulation, priorities and hierarchy. In this work we present a formal semantics for timed SFCs, which belong to the class of linear hybrid systems.

Keywords: programmable logic controllers, linear hybrid systems, sequential function charts, stopwatch semantics

1. INTRODUCTION

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 performing plant management functions.

In order to cope with this ever-growing size in terms of lines of code for implementation as well as complexity for communication and operation, the IEC 61131-3 standard (IEC, 1998) defines a number of programming languages well-suited to tackle these different aspects. *Sequential function charts* is a programming language for PLCs, which aims at providing a clear understanding of the possibly interwoven program parts. SFCs are graphical high-level notations for programs which take over ideas from *Petri nets* and *Grafcet* (David and Alla, 1992). They allow the decomposition and structuring of program parts including interesting concepts such as parallelism, activity manipulation and hierarchy. Moreover, there is the notion of time and timers, which allows to test and reason about the amount of time a program part has been active and to start program parts after a delay of time or for some limited time only. All these features can, however, again aggravate the understanding of SFCs.

Although described in IEC 61131-3, the standard leaves a lot of questions open concerning the semantics of SFCs or provides ambiguous answers as discussed later. The goal of this work is to define and, hence, clarify the semantics of timed SFCs. This serves as a basis for any discussion on SFC concepts as well as formal verification approaches in the future. The formal syntax and semantics for SFCs we present in this work comprises all the aforementioned features. In particular we focus on time and timed behavior. We show that the implementations of SFC

[1] This work has been partially supported by the German Research Council (DFG) under grant LA 1012/6-1.
[2] We thank VERIMAG, Grenoble, for support.
[3] We thank SRI International, Menlo Park, for support.

control programs are in fact (linear) hybrid systems, and we define the semantics of such systems by utilizing stopwatches (Cassez and Larsen, 2000).

The remainder of this work is organized as follows: In Section 2 we briefly introduce timed SFCs and point out the major features as well as ambiguities and open questions. Afterwards, in Section 3 we define a unifying syntax and semantics for SFCs. In Section 4 possibilities and approaches to the verification of SFCs are pointed out briefly, and we conclude with a discussion on related work and future directions in the application of formal methods to PLCs.

2. MOTIVATING A FORMAL SEMANTICS

The standard for PLC programming languages (IEC, 1998) provides intuitive and informal semantics for the execution of PLC programs. This allows for a wide range of vendor-specific implementations, but lacks a precise definition in case things are not obvious from an intuitive point of view. In this section, we point out some semantical ambiguities of the SFC programming language which need to be clarified by giving a precise semantics (the topic of Section 3).

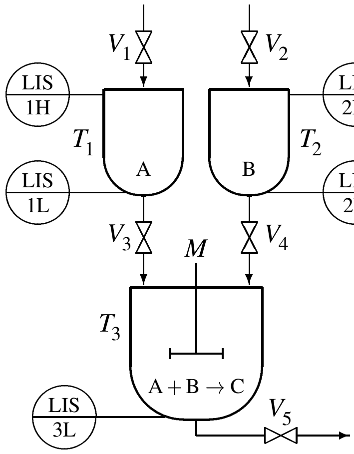

Fig. 1. Example: structure of the batch plant.

As a running example we use a simplified part of a chemical batch plant at the Process Control Laboratory of the University of Dortmund (Bauer *et al.*, 2000). Figure 1 depicts the structure of that part. Its purpose is to produce a substance C in reactor T_3 by mixing the raw materials A and B provided by the buffer tanks T_1 and T_2. The production is controlled through valves $V_1 \dots V_5$ and the stirrer motor M. Information about the liquid levels is obtained from five level sensors, where the sensors labeled with LISxL, x $\in \{1,2,3\}$ give the value *true*, if the respective tank is empty and $false$ otherwise. Sensors LISxH, x $\in \{1,2\}$ give the value $true$ if a desired level is reached.

Figure 2 shows the SFC program used to control the production process. Three tasks are running in parallel: the delivery of raw material A to T_1 through V_1, the delivery of raw material B to T_2 through V_2, and the production of C in T_3 (filling through V_3 and V_4, stirring with M, emptying through V_5). These tasks are performed in the three parallel branches of the SFC shown on the left. Since the production task is a bit more involved, it is given as an individual SFC a_3.

The basic syntactic elements of SFCs are *steps* ($s_0 \dots s_{13}$ in the example), *transitions* labeled with *guards*, and *action blocks* (consisting of an *action qualifier* and an *action name*) associated with a step. Each action name is associated with a Boolean variable (shown in quotes next to the action block), with another SFC (a_3 in the example), or with a program in another PLC language. Each SFC has an *initial step*, indicated by a double bar, such as s_0 and s_{10} in the example.

Guards are Boolean expressions over variables, where s_i.X denotes that step s_i is active and s_i.T is the time that s_i has been active since its last activation. Whenever a step is active, the qualifier of each action block associated to that step determines if the respective action will be executed. The qualifier `N` ("non-stored") activates the action as long as the step is active, `S` ("stored") activates it until an `R` qualifier resets it. The `P0` qualifier ("pulse, falling edge") activates the action for one cycle after the step is deactivated, whereas `P1` ("pulse, rising edge") activates it for one cycle whenever the step is activated. The `L` qualifier behaves like `N`, but execution is limited to the given duration, and `D` delays for the given time before the action gets executed. There are also `DS`, `SD` and `LD` qualifiers, which are explained later.

Characteristic for a PLC is its cyclic execution. A cycle consists of reading inputs (in our example: the Boolean variables for the sensor values and the "start" command are updated), doing some computation (executing actions, taking transitions), and an output phase (e.g., sending commands to the valves). In contrast to the "maximal progress" semantics for the *Grafcet* language, only one set of transitions is taken in one cycle ("lock step" semantics).

In the following we point out some ambiguities which are not resolved by the standard.

2.1 *Execution order of actions*

In which order are actions of parallel steps executed? Is there any order at all or do we have nondeterminism? The execution order is also not clear if we have more than one action associated to the same step. Are they executed from top to bottom or according to a different rule?

The order of execution makes a difference if two or more actions modify the same variable or if actions read the value of a variable that is modified by another action. We refer to such actions as *conflicting actions.* The standard does not give a clear rule for these situations. A comparison of different programming tools

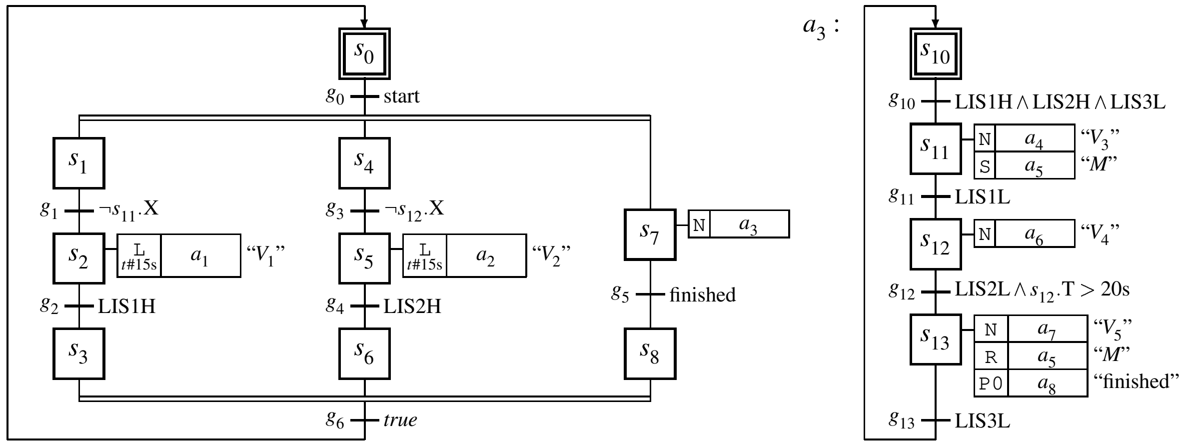

Fig. 2. Example: the control program for the batch plant.

for SFCs (Bauer and Treseler, 2001) showed that the tools have implemented different orders, ranging from the alphabetical order of action names to the order given by the graphical position in the SFC diagrams.

2.2 *Hierarchy*

As the standard allows an action to be another SFC (like a_3), a hierarchical structure of SFCs is possible. However, the standard does not define a clear semantical concept of hierarchy, thus leaving a lot of aspects undefined. It is clear that the SFC a_3 becomes inactive if step s_7 is not longer active. But does a_3 take another transition during the cycle in which s_7 is left? And what happens if s_7 is entered later and a_3 becomes active again? Does a_3 always restart at its initial step s_{10} or is there notion of history and the last active step of a_3 becomes activated again? As the standard does not provide a rule, we define that SFCs have a notion of history.

2.3 *Timed Qualifiers*

Apart from the timed qualifier `L`, it is also possible to delay the activation of an action for a certain time after the step has become active using the `D` qualifier. The `L` and `D` qualifiers can be combined with `S` yielding `DS`, `SD` and `SL`. An action associated with `DS` will only be activated if the delay time is reached *before* the step is left, whereas an `SD` action always becomes active after the elapsed time, independent of the step activity. Similar, an `SL` action is (in contrast to `L`) *always* activated for T time units. Although the action qualifier concept is defined using well-understood function blocks, the definitions in the standard still are ambiguous. For example, assume an SFC which calls an `SD` action in one step and in the next step calls the action again with the `SD` qualifier but with a different delay time. Assume that the activity time of the first step has been shorter than the first delay time, i.e. the action is not yet activated when it is called in the next step with a different delay time. Which delay time now is relevant?

These ambiguities clearly show that there is a need for an SFC semantics which gives answers to these open questions.

3. A HYBRID SEMANTICS FOR SFCS

In the following we will describe the basic elements of a formal syntax and semantics for timed SFCs. We will omit some technical details; for untimed SFCs, a more technical description can be found in (Bauer and Huuck, 2002). Important features of this semantics are *orders* on actions and transitions and the introduction of *clocks* and *stopwatches* to describe timed SFCs.

3.1 *Syntax of SFCs*

SFCs (or PLC programs in general) have different types of variables, such as input variables, output variables, and local variables, i.e. variables, that cannot be accessed from outside of the SFC. The values of the variables may belong to different data types such as Boolean or integer. The valid variable and data types are regulated by the standard, already mentioned above. We abstract here from these different variable and data types simply saying an SFC has variables which may have different values. To describe the values of all variables we use the notion of a *state* σ, which is a function assigning a value to each variable.

A state (i.e., the values of the variables) can be modified by *state transformations.* The standard defines different types of programming languages 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.

To associate action blocks to steps, we introduce a function called *action labelling function*, which gives for each step the action blocks assigned to it. For action a_1 associated to s_2 in our example, we have $block(s_2) = \{(\mathtt{L}, t\#10\mathrm{s}, a_1)\}$.

A *transition* is defined by at least one source step, one guard, and at least one target step. E.g., the transition opening the parallel branch in our example is given by $(\{s_0\}, g_0, \{s_1, s_4, s_7\})$.

A *guard* is a Boolean expression describing a set of possible values for each variable, a set of active steps and a set of possible time intervals of step durations, since we allow guards to reason about variables and step activities and the time a step has been active.

In order to cope with different execution orders of actions realized in different tools (see Section 2.1) we introduce an *order on actions*, $\sqsubset$. The order is partial, as we only need to give the execution order for conflicting actions, e.g., actions which share variables. This allows us to adapt to different execution orders realized by different tools.

Additionally, we have a partial *order on conflicting transitions*, $\prec$. For transitions, conflicting means that more than one transition starting at the same step is enabled.

To reason about timed SFCs we introduce the notions of *clocks* and *stopwatches*. A stopwatch θ_s has a dynamical behavior given by either $f(\theta_s) = \dot{\theta}_s = 1$ if the stopwatch is running, or $f(\theta_s) = \dot{\theta}_s = 0$ if it is stopped, whereas a clock θ_a cannot be stopped, i.e., the dynamical behavior is always $f(\theta_a) = \dot{\theta}_a = 1$. Both can be reset to zero. To describe the values of all stopwatches and clocks we use the notion of a *clock evaluation* ν, which is a function assigning a value to each stopwatch or clock.

As guards may reason about the time a step is active or has been active before, each step needs a stopwatch which is reset to zero when the step is entered, runs when the step is active, and is stopped when the step is deactivated. Second, we need stopwatches because we define that hierarchically nested SFCs have history, i.e., after deactivation nested SFCs are activated in the step the have been last. The notion of history includes that the SFCs remember the values of their clocks at the point of deactivation if they are activated again.

Additionally we need a clock for those actions which are associated with an `SD` or `SL` qualifier, because these actions might be deactived (`SL`) or activated (`SD`) independently from a step activity. Clocks are sufficient, because according to the standard, guards cannot access the time an action has been active, and therefore we do not need to memorize the activity time after the action has been deactivated.

An SFC is then defined as follows:

Definition 1. (SFC). A *sequential function chart* (SFC) $\mathscr{S} = (X, \Theta, S, s_0, G, T, A, block, \sqsubset, \prec)$ consists of:

- A finite set X of *variables*.
- A finite set Θ of *stopwatches* and *clocks*.
- A finite set S of *steps* s_i.
- An *initial step* $s_0 \in S$.
- A finite set G of *guards* g_i.
- A finite set T of *transitions* t_i.
- A finite set A of *actions* a_i.
- An *action labelling function* called *block* assigning a set of *action blocks* to each step.
- A *partial order on actions* $\sqsubset$ to define the execution order of conflicting actions.
- A *partial order on transitions* $\prec$ to determine priorities on conflicting transitions.

3.2 *Semantics of SFCs*

In this section we provide an operational semantics for SFCs. First of all, let us introduce some notions. It is crucial to distinguish between *ready steps* and *active steps.* Active steps are the ones control resides in and their actions will be performed. On the other hand, there might be steps where control resides in, but their actions will not be performed. These steps belong to nested SFCs which are currently not activated, i.e., there is no action active which points to this SFC. Control is “waiting” there to resume. We call all steps where control resides in *ready steps*. Hence, each active step is also a ready step, but the converse does of course not hold.

Moreover, *active actions* are actions which will potentially be executed in the current SFC cycle. This means, unless there is no matching reset action these actions will be performed. *Stored actions* are the ones which have been tagged by an `S` qualifier and potentially keep on being active outside their step of activation.

The global state of an SFC (including all its nested SFCs) is given by the values of all its variables, the evaluation of clocks and stopwatches, the sets of active and ready steps and the sets of active and stored actions. In addition to stored actions we also have to remember *stored delayed actions* associated to a step with an `SD` qualifier, which potentially need to be activated after the corresponding step has been deactivated and *stored limited actions* indicated by the `SL` qualifier, which potentially have to be executed for a certain time after the activating step has been deactivated.

We describe the global state of an SFC $\mathscr{S}$ by a configuration, given by:

Definition 2. (Configuration). A *configuration* c of $\mathscr{S}$ is an 8-tuple $(\sigma, \nu, readyS, activeS, activeA, storedA, storedDA, storedLA)$, where

- σ is the state of the variables,
- ν is the evaluation of clocks and stopwatches,
- $readyS \subseteq \bar{S}$ is the set of ready steps,
- $activeS \subseteq \bar{S}$ is the set of active steps,

- *activeA* $\subseteq \bar{A}$ is the set of active actions,
- *storedA* $\subseteq \bar{A}$ is the set of stored actions,
- *storedDA* $\subseteq \bar{A}$ is the set of stored delayed actions, and
- *storedLA* $\subseteq \bar{A}$ is the set of stored limited actions.

We use the notions $\bar{S}$ and $\bar{A}$ to denote the sets of all steps or actions of an SFC including the sets of steps/actions of its nested SFCs.

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

(1) Get new input from the environment and store the information into the state of the variables σ.
(2) Execute the SFC program, i.e., determine the new SFC configuration c'.
(3) Send the outputs to the environment by extracting the required information from the new state σ'.

The SFC program *executions* in (2) are defined by means of configuration changes within a cycle. We describe the change of configuration by associating a transition system to an SFC. A transition $c \longrightarrow c'$ in this transition system is computed by the following algorithm:

(1) Determine the new state σ' by executing all actions in *activeA* except for those which are SFCs. Conflicting actions have to be executed in accordance with the order $\sqsubset$.
(2) Determine *readyS'*. To do so, determine the set $T_{enabled}$ of transitions with source steps in *activeS* and guards holding for the current configuration. Then, *readyS'* is given by joining *readyS* with the sets of target steps of the transitions $t \in T_{ready}$ for which no other transition $t_1 \in T_{enabled}$ with higher priority exists, and removing the source steps of these transitions.
(3) Determine *activeS'*, *activeA'*, *storedA'*, *storedDA'* and *storedLA'*. The new sets are computed recursively on the structure of the SFC using an auxiliary function. The function recursively searches the top-level SFC and the hierarchically nested ones to find out if an action activated on a higher level is reset within an hierarchical lower active SFC. An algorithm for the computation of the untimed sets can be found in (Bauer and Huuck, 2002).
(4) Stopwatches and clocks are reset to zero, if one of the following holds:
    - The stopwatch θ_s belongs to a step s which has been activated in this cycle, i.e., $s \in$ *activeS'* $\setminus$ *activeS*.
    - The clock θ_a belongs to an action a which has been activated "stored delayed" in this cycle, i.e., $a \in$ *storedDA'* $\setminus$ *storedDA*.
    - The clock θ_a belongs to an action a which has been activated "stored limited" in this cycle, i.e., $a \in$ *storedLA'* $\setminus$ *storedLA*.

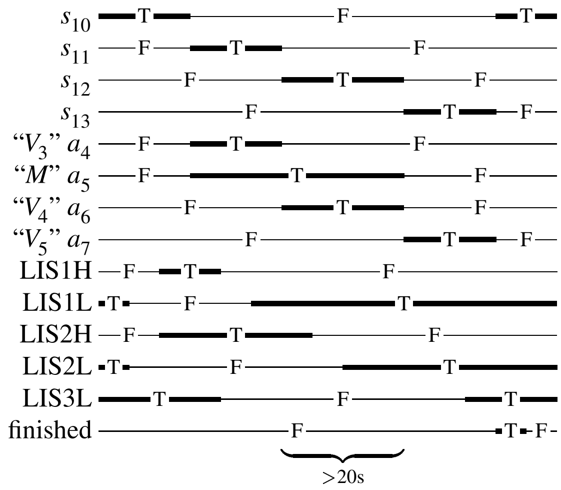

Fig. 3. Example: a fragment of a run.

An execution sequence $\langle c_0 \longrightarrow c_1 \longrightarrow c_2, \ldots \rangle$ is called *run*. The operational semantics of an SFC $\mathscr{S}$ is given by the set of runs of its associated transition system.

3.3 *Example*

Figure 3 shows a part of a run of the example shown in Figure 2. This fragment shows the evolution of the Boolean variables and step/action activities used in the nested SFC s_3 during its active phase. Note that the values for the actions a_4, a_6, and a_7 are identical to the activity of the steps they are associated with, since they are labeled with the `N` qualifier, whereas a_5 is active whenever s_{11} or s_{12} is active, since the `S` and `R` qualifiers are used. Also note that g_{12} keeps s_{12} and a_6 active for at least 20 seconds.

Such diagrams can be helpful for the manual analysis of PLC programs. Approaches for machine-supported analysis are briefly discussed in the next section.

4. CONCLUSIONS

In this work we presented an approach for a formal semantics of SFCs in order to clarify and unify the different interpretations by different PLC vendors and to use it as a formal basis for verification. In particular, this is the first work comprising all the mentioned concepts of SFCs including time. Moreover, it can cope with implementation dependent aspects, which are not well defined in the standard, such as the execution order of actions and transitions. The parameterization of the semantics allows to adjust to different interpretations of the standard and, hence, to support various available commercial tools.

There are several other approaches to define a formal framework for SFCs (Jiang and Holding, 1996), (Anderson and Tourlas, 1998), (Bornot *et al.*, 2000). However, all of them tackle restricted subclasses only and none of them defines a semantics for timed SFCs.

Future work will focus on proving the correctness of timed SFCs based on the given semantics. Currently, there exist a few approaches to prove the correctness of SFCs. However, most of them just consider Grafcet-like models or very simplistic SFCs only, e.g., (Lampérière and Lesage, 2000). A comprehensive model is studied in (Bauer and Huuck, 2001), but as the previous ones it does not take quantitative time into account. A timed approach is proposed in (L'Her *et al.*, 1998), but again for SFCs, where most features are not covered.

Hence, there is still no verification method for timed SFCs with timed qualifiers and all its distinct features. Since we deal in fact with hybrid systems this is not a simple task. As model-checking techniques do not scale well in particular for timed models, there is at least the need to find some ways to decompose SFCs. A natural decomposition into their sub-SFCs, i.e., according to the hierarchy, seems not promising, since the semantics is not layer-wise compositional, due to the action qualifier concept. However, different abstraction and deductive techniques might be worth to examine as well as how to combine them with existing automatic exploration techniques.

5. REFERENCES

Anderson, S. and K. Tourlas (1998). Design for proof: An approach to the design of domain-specific languages. *Formal Aspects of Computing* **10**(5-6), 452–468.

Bauer, N. and H. Treseler (2001). Vergleich der Semantik der Ablaufsprache nach IEC 61131-3 in unterschiedlichen Programmierwerkzeugen. In: *GMA-Kongress 2001*. Vol. 1608 of *VDI-Berichte*. VDI-Verlag. pp. 135 –142.

Bauer, N. and R. Huuck (2001). Towards automatic verification of embedded control software. In: *Asian Pacific Conference on Quality Software*.

Bauer, N. and R. Huuck (2002). A parameterized semantics for sequential function charts. Accepted for ETAPS02 Satellite Workshop on Semantic Foundations of Engineering Design Languages (SFEDL).

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

Bornot, S., R. Huuck, Y. Lakhnech and B. Lukoschus (2000). An abstract model for sequential function charts. In: *Discrete Event Systems* (R. Boel and G. Stremersch, Eds.). Kluwer Academic Publishers. pp. 255–264.

Cassez, F. and K. Larsen (2000). The impressive power of stopwatches. In: *International Conference on Concurrency Theory*. pp. 138–152.

David, R. and H. Alla (1992). *Petri Nets & Grafcet*. Prentice Hall.

IEC (1998). *Programmable Controllers – Programming Languages, IEC 61131-3*. second ed. Committee draft.

Jiang, J. and D.J. Holding (1996). The formalisation and analysis of sequential function charts using a Petri net approach. In: *Proceedings of 13th World Congress of IFAC*. pp. 513–518.

Lampérière, S. and J.-J. Lesage (2000). Formal verification of the sequential part of PLC programs. In: *Discrete Event Systems* (R. Boel and G. Stremersch, Eds.). Kluwer Academic Publishers. pp. 247–254.

L'Her, D., J.L. Scharbarg, P. Le Parc and L. Marcé (1998). Proving sequential function chart programs using automata. *Lecture Notes in Computer Science* **1660**, 149 – 163.