

Symbolic Model-Based Testing for Industrial Automation Software

Sabrina von Styp¹ and Liyong Yu²

¹ Software Modeling and Verification
Department of Computer Science

² Chair of Process Control Engineering
RWTH Aachen University Germany

Abstract. In industrial automation software controls systems whose failure can be critical and expensive. Testing this software is very crucial but so far done manually, an expensive and not very thorough method. Model-based testing is an emerging concept in computer science for automatically testing a real implementation. It uses a formal specification describing the system behaviour. This specification is the blue print against which an implementation is tested. This paper presents how to use model-based testing in industrial automation. In detail it shows how the known concepts such as sequential function charts, used in industrial automation to describe a system, can be translated to a format that is required for model-based testing, including an automatic derivation of test-cases and its execution. A concrete case study illustrates the strength of this approach.

1 Introduction

Software testing is crucial especially for safety critical systems such as controllers in industrial automation. Unexpected execution results can be caused by semantical inconsistencies among requirements defined by function designers, comprehension on part of human programmers, and runtime characteristics of hardware. Considering the last aspect, automation functions are generally implemented on Programmable Logic Controllers (PLCs) or PLC-based Decentralised Control Systems (DCSs). PLC programs are characterised by their cyclic execution. Single PLC programs are cyclically executed without supporting multitasking, meaning that components of a program (e.g. steps and transitions of the same procedure) are not executed simultaneously, but sequentially. As a result, execution results can be influenced by scheduling of program parts, for instance function blocks within a function block network, actions of procedure step, steps and transitions of a procedure. Discussions with more details can be found in [3],[11] and [29]. PLCs and DCSs often control plants whose failure can be expensive and dangerous. Thorough testing therefore is crucial but also time consuming and expensive.

One attractive method to automate the test process, and therefore reduce time and cost, is *model-based testing* [6]. Model-based testing is a so-called black

box testing technique, which automatically derives the test-cases from a formal specification describing the desired behaviour of the implementation under test (*IUT*). These test-cases are then executed against the real *IUT* by stimulating the *IUT* and observing its output. The underlying notion of correctness of an implementation, with respect to its specification, allows to show that testing can not yield false alarms.

One well-known theory for model-based testing is the *ioco framework* [26], where the specification is given as a labelled transition system (*LTS*) with input and output actions. The conformance relation, called *ioco*, describes that after a trace an implementation may only produce outputs or be silent if the same outputs or silence are also produced by its specification. Several tools exist for the derivation of *ioco* test-cases, e.g. JTorX [5], AGEDIS TOOLSET [10] and TGV [18]. The disadvantage of using an *LTS* as a specification is the state space explosion that occurs when using variables with larger domains, i.e. integers. To avoid this problem, the *sioco* framework [7,8], using symbolic transition systems (*STS*) as specification, has been developed. *STS*s allow a symbolic representation of data as well as a data dependent control flow, and therefore avoid the problem of infinite branching and infinite state space. The notion of symbolic testing has been implemented in the test tool JTorX [5].

There exists two approaches [16,17] to apply model-based testing in industrial automation engineering. Both approaches use the modelling language UML and only work for special application cases. Even though symbolic model-based testing has been successfully applied in many areas, to the best of our knowledge it never has been used in industrial automation engineering using sequential functional charts (*SFC*s), a well-known formalism which formally describes the steps a system can execute and under which circumstances it has to change into a different step. Bauer et al. [2] have successfully applied formal verification, precisely model-checking, in industrial automation. They used *SFC* as modelling language and provided a translation from *SFC*s to timed automata and used the tool UPPAAL [27] for model checking. Their approach can be used to verify the correctness of the provided formal specification, but does not verify if the real implementation is correct to its specification. For model-based testing only observable in- and outputs are of interest and therefore the approach of [2] is not suitable for model-based testing. There is also no notion of handling variables symbolically, leading to the state space explosion problem when handling variables domains such as reals or integers. To circumvent this problem Bornot et al. [4] used symbolic modelling verification [19] for applying model-checking on *SFC*. Their work also considers internal behaviour of the *SFC* and is therefore not suitable for model-based testing. Hence, we provide a new translation from *SFC*s to symbolic transition systems, where only observable behaviour is considered. In previous work [25], a translation of *SFC* to *LTS* has been introduced. This approach encounters a state space explosion when testing larger variable domains. To circumvent this state space explosion this paper provides a translation from *SFC* to *STS* and presents two case studies, showing how to do model-based testing with the *sioco* framework in industrial automation. The case

studies demonstrate that real errors such as an imprecision in representing floats can be found by our approach. The first case study of this paper was inspired by previous work [25].

First, this paper gives an introduction on functional description methods in industrial automation. Then the underlying theory of the *sioco* framework is introduced. Section four presents the translation of *SFC* to *STS*. Section five describes the environment used for the case studies. Section six presents two case studies and demonstrates how to apply the introduced concept in practice. Finally, the paper is concluded in the last section.

2 Sequential Function Charts as Functional Description

The clear description of requested functions, especially the internal discrete logic (e.g. state machines and sequential procedures), is a logical prerequisite to the test of PLC-implementations. In todays industrial automation, there still exists no standardised description method. Discrete logics on PLCs can be described in various forms, for instance in Petri nets, UML/Statecharts or Procedure Function Charts (PFCs)[14]. Although existing methods with different expressive power can well represent functions in specific application domains, most of them do not consider execution semantics on PLCs. In order to ensure a consistent implementation, many automatic code generators have been presented in a series of research works. However, mapping rules that have been taken are usually developer-specific, and are often only suitable for hardware from certain vendors.

In the landscape of description methods for discrete logics, *SFC* is an approach that provides standardised graphical description and vendor-neutral implementation technique at the same time. *SFC* follows the specification language *GRAFCET* [12], which is based on Petri nets. *SFC* is proven to have the potential of being developed to a general description method [29][28].

Sequential functions of any complexity can be precisely described by simple graphical elements: steps and transitions between steps. All *SFC*s begin with an initial step (marked with a double boundary line), and they can either end with a final step or jump back to a previous step at the end of the chain. The former design is normally used to describe chemical production procedures, which only need to be worked through once. In turn, the latter one can describe a permanently active state machine. A discrete function described in *SFC* language can be directly implemented in a PLC. However, there are also systems that do not support graphical *SFC* programming, or whose implementation should not be programmed in *SFC*. For instance, the function block for a single motor controller discussed later in this paper is coded in most automation systems by using a textual programming language. Nevertheless in these cases, the internal execution progress and state machines of the encapsulated function block can also be intuitively and exactly described by using *SFC*. In this way, the programming engineer can exactly understand requirements given by the designer with the help of the *SFC* graphic; on the other hand, if an existing implementation needs to be optimised, its *SFC* description can help the engineer to understand

the working principle and define solutions. In this paper, we show how *SFC* descriptions can also be used for automatic testing by automatically deriving and executing test-cases from them.

As a programming language, specifications of SFC in IEC 61131-3 [13] are not suitable to be directly applied as a description language. Improving SFC is part of a further research work [29], which intends to develop a general description method for discrete functions in a wide range of automation domains. For this paper, we consider a simplified SFC with the following features:

- Inspired by UML/Statechart, only three alternative action qualifiers for controlling action occurrences are applied: *P1*, *N* and *P0*. Their semantics correspond respectively to entry, do and exit.
- Actions of the same steps are executed sequentially from top to bottom (cf. the example SFC in Figure 9).
- Simultaneous sequences are not allowed. Only one step can be activated, and it represents the overall execution progress of the *SFC*.
- Only one alternative sequence can be chosen (compare Figure 9). In case of branching, transitions are evaluated from left to right.
- The *SFC* precedes its execution by the sequence in which the transition is evaluated as switchable at first.

3 Testing with Symbolic Transition Systems

Model-based testing is a well known technique for automatically generating test-cases and its execution. It is based on a conformance relation, e.g. *ioco* [26] for labelled transition systems and *sioco* [7,8] for symbolic transition systems (*STS*), which formally defines when an implementation is correct with respect to a given specification. For the conformance relation it is assumed that the real *IUT* could theoretically be described by a transition system, see Figure 1.

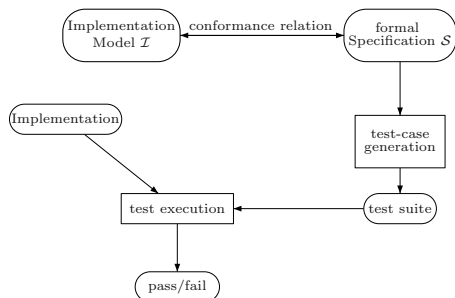


Fig. 1. Overview of model-based testing

The conformance relation formally describes when an implementation is correct with respect to its specification. The *ioco*-relation basically requires that an implementation may only produce an output if the specification can produce the same output after executing the same trace. Based on this, the test-cases can automatically be generated from the specification and build a so-called test suite. These test-cases are used to test the real *IUT*. The verdict is either fail together with a trace leading to the error, or pass, provided the state space is finite. In case of an infinite state space, which mostly is the case, the implementation is tested until an error is found. Although this procedure is not complete, the advantage of model-based testing over manual testing is, that the implementation can be tested without any human interaction. After the execution of a certain amount of test-cases it could

also be considered to test a copy of the implementation in the background while already using the implementation, to find additional errors.

Most controllers in industrial automation use variables of large, or even infinite domains such as integer or real numbers. In order to allow these variable domains and a data control flow without state space explosion, symbolic testing is indispensable. For symbolic model-based testing, symbolic transition systems (STSs) are used. STSs introduced by Frantzen et al. [7,8], are labelled transition systems extended with *in- and output gates* omitting interaction variables, *guards* over variables mostly written in first-order-logic (FO-logic) and *variable updates*, which manipulate the values of variables while performing transitions.

3.1 Symbolic Transition System

A symbolic transition system consists of locations including a initial location, interaction variables, location variables, in- and output gates containing interaction variables and an edge relation. Interaction variables are used in in- and output gates to transmit values. Location variables are internal variables and can be used to store the value of interaction variables, which can be edited from outside the system.

Definition 1 (Symbolic Transition System). *A symbolic transition system STS is a tuple $\mathcal{S} = (L, l_0, \mathcal{V}, \mathcal{I}, \mathcal{G}, \rightarrow)$, where*

- L is a finite set of locations where $l_0 \in L$ is the initial location
- \mathcal{V} is a finite set of location variables
- \mathcal{I} is a finite set of interaction variables, where $\mathcal{I} \cup \mathcal{V} = \text{Var}$ and $\mathcal{I} \cap \mathcal{V} = \emptyset$
- $\mathcal{G} = \mathcal{G}_{out} \cup \mathcal{G}_{in}$ is a finite set of gates, where every gate has a tuple of interaction variables of certain length. \mathcal{G}_{out} is the set of output and \mathcal{G}_{in} the set of input gates, where $\mathcal{G}_{out} \cap \mathcal{G}_{in} = \emptyset$;
- $\rightarrow \subseteq L \times \mathcal{G} \times \mathfrak{F}(\mathcal{V}) \times \bigcup_{V \subseteq \mathcal{V}} \mathfrak{T}(\text{Var})^V \times L$ is the edge relation.

The notation $\mathfrak{F}(\text{Var})$ denotes the set of FO-formulas over all interaction- and location variables. $\mathfrak{T}(\text{Var})^V$ denotes the set of mappings of interaction variable to variables of Var . As usual we write $l \xrightarrow{g, \varphi, \rho} l'$ to denote an element of $(l, g, \varphi, \rho, l') \in \rightarrow$, where g is a gate with interaction variables, φ a guard over variables and ρ a mapping of interaction variables to location variables.

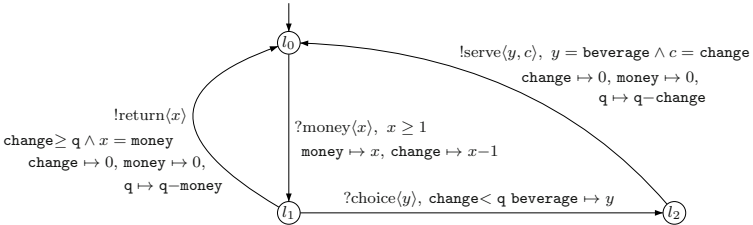


Fig. 2. A sample STS-beverage-vending-machine

Example 1. *Figure 2 shows an STS with the locations l_0, l_1, l_2 modelling a beverage vending machine. The machine expects money in bills (parameter x of input $?money$), a choice for a beverage and returns change (parameter c of*

output !serve). If there is not enough change in the machine on a bill, the bill is returned (output !return). Otherwise, the user can choose a beverage for the cost of one unit, the change is returned and the beverage served (parameter y of output !serve). Parameters x, y, c are so-called interaction variables. Variables q , change, beverage, money are so-called location variables. Interaction variables represent the possible values that can be passed during input and output.

The symbolic trace semantics is defined by Frantzen *et al.* [8] and expressed by transition relation $l \xrightarrow{\sigma, \varphi, \rho} l'$ where σ is a sequence of gates that are executed on the trace from l to l' , φ is a conjunction of constraints over variables that need to be satisfied to reach location l' , and ρ is a concatenation of variable mappings denoting the possible variable valuation after l' has been reached. Frantzen *et al.* [8] also define a conformance relation *sioco* describing when an implementation is correct to a specification. The conformance relation is based on *ioco*, introduced by Tretmans [26], and describes that an implementation can only produce an output, after executing a trace σ , if the specification can produce the same output after executing σ . Quiescence of the implementation is only allowed if the specification is also quiescent after executing the same trace. Going into further details would go beyond the scope of this paper and we refer to Frantzen *et al.* [8] for more details.

3.2 Testing with JTorX

The test-tool JTorX [5] is a platform-independent tool for model-based testing. It automatically derives test-cases from a given specification using the *sioco* relation. It therefore considers the traces in the specification and instantiates the input variables with concrete values regarding to the corresponding guard. Outputs are verified by checking if the output gate can also be mimicked by the specification and if the variable values satisfy the corresponding guard.

The specification of the program is provided to JTorX in XML format. In order to communicate with the *IUT*, JTorX supports standard input and output as well as the network protocol TCP/IP. JTorX requires these inputs and outputs to have the same format as the one given by the specification. Therefore, it is common to provide an adaptor that coordinates the different input and output formats. The test-case generation in JTorX happens on-the-fly, meaning that test-case generation and execution are done at the same time. Only the next steps that are needed are computed, and the information, which has already been traced, is stored in a log file and can be executed again if requested later. For modelling the *STS*, JTorX uses the *STSsimulator*-framework [22].

4 Translation of *SFC* to *STS*

The previous section shows that for model-based testing with data the specification has to be a symbolic transition system. However, in industrial automation *SFCs* are used to formally describe the system behaviour. Therefore, this section addresses the translation of *SFCs* into *STSs*. Since model-based testing is

a form of black-box testing, the translation only focuses on observable actions, such as inputs and outputs. In an *SFC*, variables are changed by the system when executing a step, while requirements for a transition are usually changed by some external components. As the transitions control which step is executed, and therefore how variables change, the variables enabling the transition are considered as input variables and the variables changed by the implementation, when executing a step, are considered as output variables. In theory it is possible that variables can be used for input and for output, as allowed in the following theory, but in practice it might be necessary to prohibit this.

4.1 Formal Description of *SFC*

In order to provide a translation from *SFC* to *STS* we first give a formal definition of *SFC*. As only observable actions are of interest, the definition is restricted to those. An *SFC* has a set of steps that can be executed and end with a set of observable variable configurations. The execution of an *SFC* is cyclic. This means every step and therefore all actions in one step are executed within one cycle. At the end of every cycle it is checked if one of the following transitions is enabled. Subsequently, the transition which is satisfied is taken and its successor step executed in the next cycle. In case more than one transition is enabled, the one with the highest order, the left most in a drawn graph, is taken.

Definition 2 (Sequential Function Charts). A *Sequential Function Chart (SFC)* is a tuple $\mathcal{S} = (S, T, \mathcal{X}, \mathcal{Y}, \mathcal{R}, \rightarrow_{SFC})$ where:

- S is a finite set of steps containing actions, e.g., set variables, over output variables
- T is a finite set of transitions containing FO-formulas over input variables as guards
- \mathcal{Y} is a finite set of output variables and \mathcal{X} is a finite set of input variables
- $\mathcal{R} \subseteq T \times T$ is a total ordering on the set T .
- $\rightarrow \subseteq S \times T \times S$ an edge relation

In a step s_k , variables y_{k1}, \dots, y_{kn} are set to certain values. Such a setting is of the form $y_{ij} := a \oplus b$, where $\oplus \in \{*, -, +, /\}$ and $a, b \in \mathcal{X} \cup \mathbb{R}$. The execution of a step s is repeated until a following transition t is enabled, written as $t \in en(s)$. A transition t_l has conditions c_{l1}, \dots, c_{lm} over input variables x_{l1}, \dots, x_{lp} . They are concatenated by disjunctions or conjunctions, where c_{ij} is of the form $a \otimes b$ with $\otimes \in \{<, >, \leq, \geq, =\}$ and a, b being arithmetic expression over the set $\mathcal{X} \cup \mathbb{R}$. If more than one transition is enabled, transition t with the highest order wrt. \prec , written as $\forall t' \in T, t' \neq t : t' \prec t \wedge en(t')$, is taken.

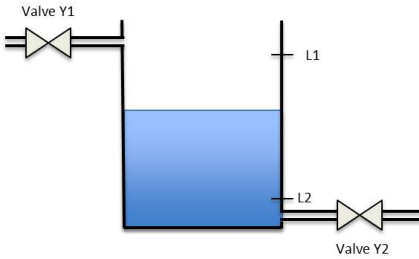


Fig. 3. Tank System

We write $\mathbf{trans}(s)$ to denote all transitions that can be enabled after step s . Let us consider a simple example. Figure 3 shows a tank system for some fluid. The input can be controlled by valve Y1 and the output by valve Y2. Is the fluid level above sensor L1, the value of L1 is true. The same holds for sensor L2 if the fluid level is below L2.

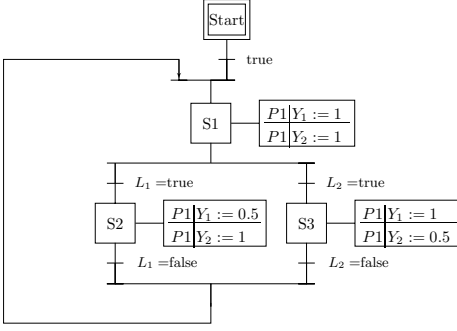


Fig. 4. SFC of tank system

both valves are 100% open. In case the fluid level is above $L1$ valve Y_1 is set to 50% and valve Y_2 to 100%. Is the fluid level to low and $L2$ is true, step $S3$ is executed.

4.2 From SFC to STS

The formal definition of an *SFC* now allows to define the translation to an *STS*. The idea is to consider the variables of a step as outputs and the ones at the transitions as input, as they allow to control the flow of the system. In an *SFC* every step is followed by at least one transition. This yields that in the *STS* every output is followed by one input, representing all possible inputs. Executing a step more than once can be accomplished if the input does not enable any guard of edges representing next steps. The only guard that then is enabled is the guard of the edge representing the previous output. Inputs have no guards, since the input values are usually changed by some external events and cannot be controlled. Every input action is followed by at least one output action. Every output action has a guard making sure the correct previous transition has been enabled before and the variable output is correct. An *STS* $\mathcal{T} = (L, l_0, \mathcal{V}, \mathcal{I}, \mathcal{G}, \rightarrow)$ obtained from an *SFC* $\mathcal{S} = (S, T, \mathcal{X}, \mathcal{Y}, \mathcal{R}, \rightarrow_{SFC})$ is defined as follows:

- $L = \{l_s | s \in S\} \cup \{\hat{l}_s | s \in S\}$ set of locations with initial location l_0 , representing the edges between location and steps of the *SFC*
- \mathcal{V} location variables
- $\mathcal{I} = \mathcal{X} \cup \mathcal{Y}$ interaction variables
- \mathcal{G} set of in- and output gates containing interaction variables
- \rightarrow a edge relation where every $s \xrightarrow{t}_{SFC} s'$ induces
 1. $l_s \xrightarrow{g, true, \rho} \hat{l}_s \xrightarrow{g', \varphi', \rho'} l_{s'}$ where
 - $g = ?in(\bar{x})$,
 - $\rho = (\bar{v}_{in} := \bar{x})$,
 - $g' = ?out(\bar{y})$,
 - $\varphi' = (\bigwedge_{t \in en(s), t \prec t'} \neg F(t')) \wedge F(t) \wedge F(s)$ and
 - $\rho' = (\bar{v}_{out} := \bar{y})$
 2. $\hat{l}_s \xrightarrow{g', \varphi'', \rho'} l_s$, where $\varphi'' = (\bigwedge_{t \in trans(s)} \neg F(t)) \wedge F(s)$

Figure 4 shows the *SFC* of the controller for the tank system. The steps of this *SFC* are $S1, \dots, S3$ and the transitions are the conditions between the steps, e.g. $L_1 = true$. On the right side of every step the boxes describe the actions that are executed if the step is active, e.g. in step $S3$ the variables $Y_1 := 1$ and $Y_2 := 0.5$ are set. $P1$ in front of the actions stands for exactly one execution of the following action, see also Section 2. In the beginning

The location l_s represents that the step s in the *SFC* has been executed and next it is checked which transition after s is enabled, represented by \hat{l}_s . We write \bar{x} to denote the vector $\bar{x} = x_1, \dots, x_n$ and $\bar{v}_{in} := \bar{x}$ to assign $v_{in_i} := x_i$. The FO-formula $F(s)$ describes the effects of executing step s . It is the conjunction of all variable settings (see Definition 2) occurring in s combined with $\bigwedge_{1 < i \leq n} y_i == v_{out_i}$ where y_i is not changed in s . The last part ensures that variables that are not changed, remain their current values.

The FO-formula $F(t)$ represents the condition that enables transition t . The second clause in the definition above models that in an *SFC*, the execution of a step is repeated until one of the transitions afterwards is enabled. In case no condition for the next output edge in the *STS* is satisfied, the *STS* goes back, with the same output as before, to location s .

Example 2.

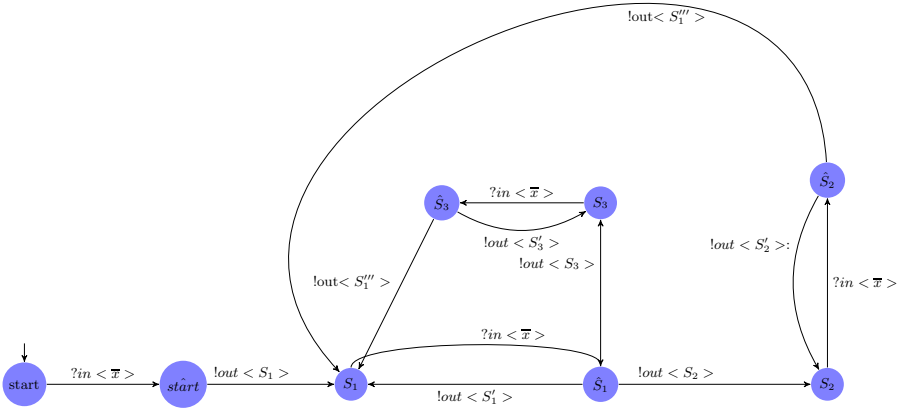


Fig. 5. *STS* obtained from the tank system *SFC*

- ?in < \bar{x} >: ?in < x_1, x_2 >, $l_{x_1} = x_1, l_{x_2} = x_2$
!out < S_1 >: !out < y_1, y_2 >, $y_1 == 1 \wedge y_2 == 1, l_{y_1} = y_1, l_{y_2} = y_2$
!out < S_1' >: !out < y_1, y_2 >, $y_1 == 1 \wedge y_2 == 1 \wedge \neg l_{x_1} == 1 \wedge \neg l_{x_2} == 1,$
 $l_{y_1} = y_1, l_{y_2} = y_2$
!out < S_2 >: !out < y_1, y_2 >, $l_{x_1} == 1 \wedge y_1 == 0.5 \wedge y_2 == 1,$
 $l_{y_1} = y_1, l_{y_2} = y_2$
!out < S_3 >: !out < y_1, y_2 >, $\neg l_{x_1} == 1 \wedge l_{x_2} == 1 \wedge Y_1 == 0 \wedge Y_2 == 1, l_{y_1} = y_1, l_{y_2} = y_2$
!out < S_2' >: !out < Y_1, Y_2 >, $l_{x_1} == 1 \wedge Y_1 == 0.5 \wedge Y_2 == 1, l_{Y_1} = Y_1, l_{Y_2} = Y_2$
!out < S_3' >: !out < Y_1, Y_2 >, $l_{x_2} == 1 \wedge Y_1 == 0 \wedge Y_2 == 1, l_{Y_1} = Y_1, l_{Y_2} = Y_2$
!out < S_1'' >: !out < Y_1, Y_2 >, $l_{x_1} == 0 \wedge Y_1 == 1 \wedge Y_2 == 1, l_{Y_1} = Y_1, l_{Y_2} = Y_2$
!out < S_1''' >: !out < Y_1, Y_2 >, $l_{x_2} == 0 \wedge Y_1 == 1 \wedge Y_2 == 1, l_{Y_1} = Y_1, l_{Y_2} = Y_2$

Figure 5 shows the *STS* obtained from the *SFC* in Figure 4. The edge from *start* to *start-hat* is an input for enabling t_0 , the first transition in the *SFC*. All input edges consist of a gate with the interaction variables x_1, x_2 , standing for the level sensors. These variables are mapped to location variables l_{x_1} and l_{x_2} . All output edges have a gate with the variables y_1, y_2 for the valves, a guard and a mapping of interaction variables to location variables. The edge from *start-hat* to S_1 is the output, namely the observable result, of step S_1 in the *SFC*. S_1 to \hat{S}_1 is an input representing possible enabling of t_1 or t_2 . If none of the guards, representing the transitions in the *SFC*, is enabled, output !out < S_1' >, leading back to location

S_1 , has to be observed. This models the behaviour of the *SFC* repeating a step if no transition afterwards is enabled. If a transition t_1 or t_2 is enabled $!out < S_{2,3} >$, depending on the previously enabled transition, is executed. Is more than one transition enabled the one with the lower index is taken, as this is the transition order. After that output the system is in state S_2 or S_3 depending which $t_{1,2}$ has been enabled. Then a repetition of the execution of the state is possible if the transition afterwards is not enabled. This is modelled by the edge $!out < S'_{2,3} >$. Is the required edge constraint satisfied the output $!out < S''_1 >$ or $!out < S'''_1 >$ can be observed. S''_1 and S'''_1 both have a guard requiring the correct execution of step S_1 in the *SFC*, but their condition regarding the previously enabled edges differs as required by the *SFC*.

5 Case-Study Set-Up

The previous section described the usage of *SFC* as a formal description language and its translation to *STS*. This is presented in the upper part of Figure 6. The formal description, given as an *SFC*, describes the desired behaviour of the implementation. The implementation itself runs in an industrial operative environment (e.g. a runtime server on a PLC), and is cyclically executed with a predefinable and constant cycle time. Industrial communication protocols like OPC UA [15] and ACPLT/KS [1] allow standardised and system-neutral access (e.g. get and set of variables) to the current datas from external clients. The cyclic execution of the implementation can be controlled by a certain variable. It allows the general starting and stopping as well as exactly one cyclic execution. The feature of executing exactly one cycle is crucial for the test case execution.

The *STS* is obtained from the *SFC* before the testing process is started. It is then provided to JTorX [5], which uses it as a basis to generate test-cases. The adapter handles the communication between JTorX and the operative environment. It receives the input from JTorX, sets the corresponding variable values while the implementation is offline and executes the implementation exactly one cycle. When the implementation is offline again, the adapter gets the values of the variables and passes them on to JTorX, which interprets them as an output by the implementation.

5.1 Operative Environment

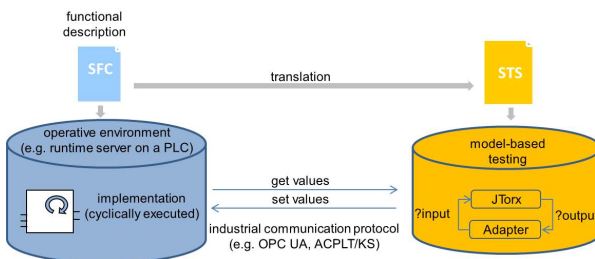


Fig. 6. Application Structure

PLCs from different vendors often provide different software and hardware structure as well as execution behaviours. In order to develop a generic testing approach without dependence on specific

execution platforms, we have applied a generic operative environment named ACPLT/OV [20]. ACPLT/OV is an open-source object management and runtime environment which permits the development of reference models and object-oriented applications that can be operated in real-time industrial automation. ACPLT/OV provides an object-oriented API for ANSI C, and allows a platform-independent evaluation of different models and execution behaviours. So far, an OV-server can be installed on industrial PCs supporting Linux and Windows and some micro controllers. An ACPLT/OV application is hosted on a server, which contains the executed program as well as object models. An OV-server can be directly applied as operative environment, but may also be set up on a conventional computer to act as an emulator of the industrial operative environment in prototyping tasks.

OV-servers offer an ACPLT/KS [1] interface for the information exchange with external servers and clients. ACPLT/KS is an open-source client/server communication protocol designed for decentralised control systems (DCS) and related applications. It uses object-oriented meta-modelling, where the predefined elements of the communication protocol (variables, domains, links, etc.) are generic, and can be used to manipulate virtually any concrete object model of a control system. ACPLT/KS clients have been developed in C++, Tcl/Tk, JavaScript and VB at present.

In our case study, every function that has to be implemented is programmed manually and encapsulated in a function block according to the IEC 61131-3 standard [13]. Internal logic of one function block is cyclically executed as far as the function block is activated (i.e. enabled). Information exchange with the environment is realised by input and output variables.

5.2 Client-Server Communication

The input and output vectors that are provided and observed – respectively – by JTorX, are given by the *SFC* specification. JTorX does not have a direct way of setting or getting variable values of the tested system, but the OV server provides the means to set and get these values exclusively through ACPLT/KS. Additionally, it has to be ensured that input variables are set at the right time on the server, and that JTorX always receives the current variable values, i.e. JTorX and the tested system have to be synchronised. This requires an adapter that sends and receives values in between JTorX and the server, and which also performs the required synchronisation by starting and stopping the IUT. The adapter is written in Java and communicates via standard input and output with JTorX, while interacting with the server through ACPLT/KS.

6 Case Studies

In order to demonstrate the practical feasibility of our approach we conducted two case studies. The first one is a function block that controls a motor. The second case study is a function block controlling a heat exchanger. Here, the variables are reals representing the temperature, and therefore symbolic testing

is inevitable. For both case studies, the translation from PLCopen XML, an XML format for IEC 61131-3 languages [21] to `.sax` was done manually. A program for automatic translation is currently developed. For more detailed information about the case studies we refer to [23].

6.1 Case Study Motor

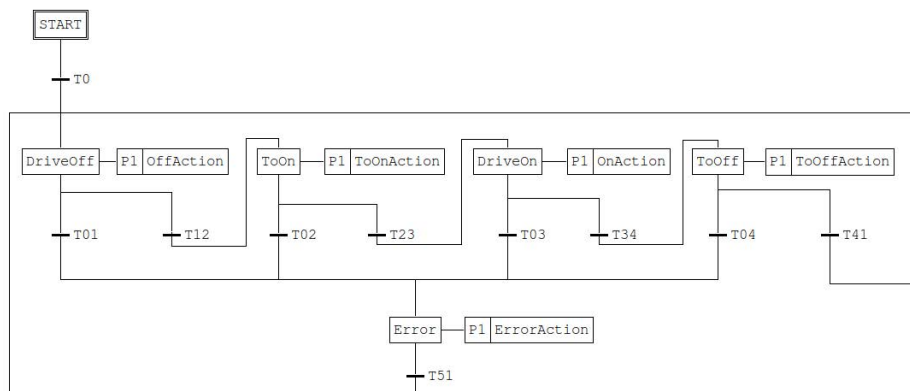


Fig. 7. SFC specification for the `simpleMotor` function block

For our case study, we chose a simple function block that controls an on/off motor with five steps. The function block has the following Boolean inputs: `Con` and `Coff` indicate that the motor should be switched on and off, respectively; `CACK` indicates that the user has acknowledged an error; and `chkbOn` indicates that the motor has confirmed that it has switched itself on, known as check back. In turn, the Boolean outputs are: `ACT` signals the motor to switch on (true) or off (false); `DriveOn` indicates that the motor is on; `DriveOff` indicates that the motor is off; and `ERR` indicates that an error has occurred.

The intended behaviour of the function block is described as follows. The motor is initially switched off, and the user may set `Con` to `TRUE` in order to start the motor. The function block then sets `ACT` to `TRUE` in order to switch the motor on, and waits for a confirmation signal with the value `TRUE` from the motor on `chkbOn`. If the confirmation signal arrives, the `DriveOn` indicator is set to `TRUE` and the function block stays in this state until the user sets `Coff` to `TRUE` in order to stop the motor. In this case, the function block sets `ACT` to `FALSE` in order to switch the motor off, and waits for a confirmation signal with the value `FALSE` from the motor on `chkbOn`. When this occurs, the function block returns to its initial state. In any case, an error state may be reached whenever an unexpected confirmation signal from the motor is received. In this case, the function block sets `ERR` to `TRUE` and stays in this state until the user acknowledges the error by setting `CACK` to `TRUE`, which clears the error indicator and returns to the function block's initial state.

The control logic for the `simpleMotor` function block has been specified by the *SFC* in Figure 7. Here, the steps `DriveOff`, `ToOn`, `DriveOn`, `ToOff` and `Error`

represent the different states of the `simpleMotor` function block, and their corresponding actions set the function block outputs to the expected values. Furthermore, the transitions evaluate the conditions for a step change which depend on the function block inputs. Based on the *SFC* specification, the `simpleMotor` function block was implemented in C. In addition to this object class, five test classes were produced as exact copies of the original class, but with manually introduced errors, with the purpose of validating the model-based testing approach.

Since *JTorX* works on *STS*, we first derive the *STS* from the *SFC* specification as defined in Section 4. The *STS* for the `simpleMotor` has 13 locations and 21 transitions.

In the first phase, the transition system and the `simpleMotor` function block were used. The system under test was executed for 100,000 steps which took 13 hours. Still this is faster than testing manually and less expensive. Most of the time is due to the tree solver used in *JTorX* for generating the test-cases from the *STS*. The same case study in [25] with concrete labels took only 190 minutes as no transition guard had to be solved. The implementation received 50,000 inputs and the same number of outputs were observed and validated by *JTorX*. In the second phase, five mutants of the implementation were tested. They were tested without knowing the error manipulation. All errors were found within two hours using *JTorX*. There were no false alarms.

6.2 Case Study Heat Exchanger

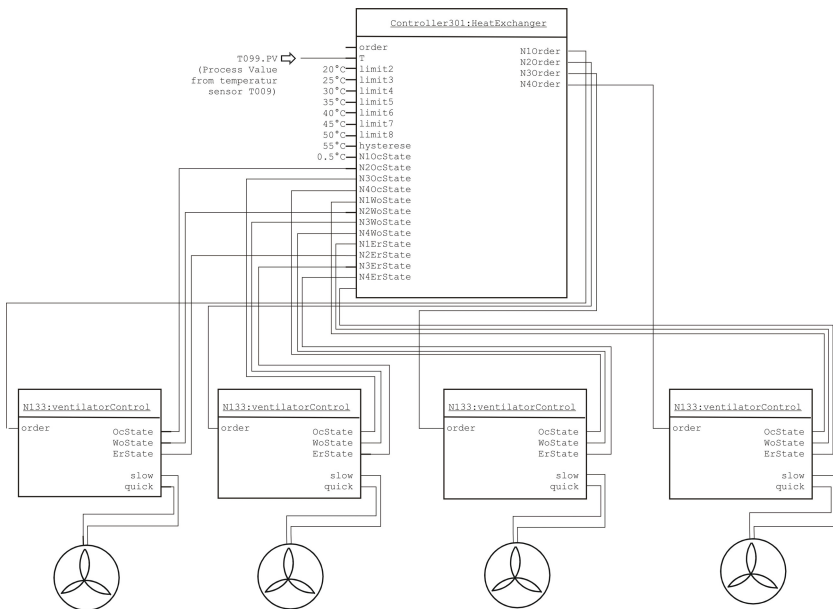


Fig. 8. Control units for the heat exchanger

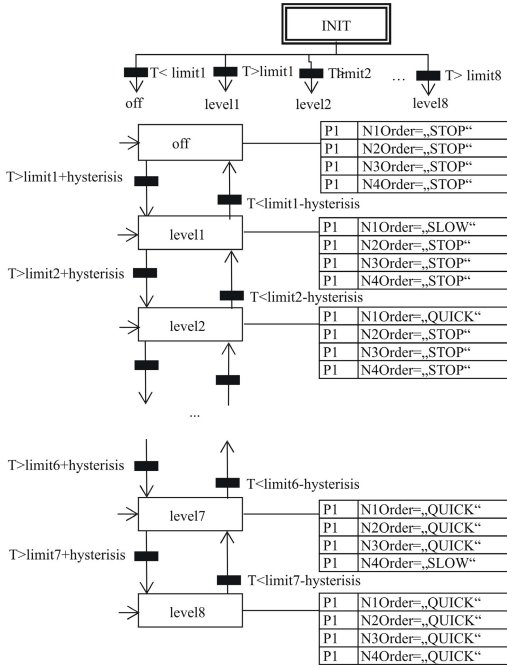


Fig. 9. SFC specification of the function block *HeatExchanger* (simplified)

industrial PC from a commercial vendor. It has about 1400 lines of code, excluding header classes and the server it is running on. During a typical production, temperature in the furnace is over 3000°C, while temperature of the cooling water should be kept under 55°C. Considering production safety, the heat exchanger should be operated in a reliable manner.

In the control program, 4 ventilator controllers and one group controller for the whole heat exchanger have been used. The group controller is implemented as a function block that realises the control logic in Figure 9. The main inputs and outputs of this block are declared as follows:

Name	data type	variable type	description
$N_x Order$	STRING	output	commando for ventilator N_x ($x=1,2,3,4$)
$N_x OcSt$	UINT	input	occupancy state of N_x (10=free, 20=hand, 30=automatic)
$N_x WoOcSt$	UINT	input	working state of N_x (10=off, 20=starting up, 30=stationary, 40=stopping)
$N_x ErSt$	UINT	input	error state of N_x (20=undifined, 20=good, 30=alarm, 40=warning)

Test-Case Execution. The *STS* obtained from the *SFC* has 22 states and 63 transitions. The highest limit in the heat exchanger is 50°C, after which all four ventilators have to run. To avoid that *JTorX* tests arbitrary high temperature values and to speed up the test process, the maximal temperature value was restricted to 100°C and was chosen randomly by *JTorX*.

Figure 8 illustrates the structure of a water cooling system for a metallurgical furnace. The core of this cooling system is a heat exchanger composed of pipelines, four identical ventilators and one temperature sensor. Every ventilator can be operated in three modes: stop, slow and quick. By applying different combinations of ventilators, the water temperature can be kept in a certain range.

The control logic of the heat exchanger is described as an SFC, see Figure 9. Altogether 8 control levels (i.e. control states) are defined. According to the measured temperature, the control logic moves to the corresponding control level and operates the ventilators differently. The control program was programmed in ANSI C and executed on an in-

Testing the heat exchanger with JTorX revealed three errors. One error was due to an imprecision in the fourth decimal after the comma when representing floats, e.g. 23.99998 was shown by the server the implementation was running on. This problem could not be fixed and such an accuracy is not needed in practice most times, so the model was changed to accept a tolerance of 0.0001 for further tests. The other two errors were errors made by the programmer and could have resulted in a system state where all ventilators were switched off even though the temperature exceeded 50°C . All observed errors were errors that were not found when intensively testing the system manually.

7 Conclusion

This paper proposed to automatically test controllers in industrial automation by using symbolic model-based testing. Due to the popularity of *SFC* in industrial automation our testing approach is based on *SFC* specification. We have implemented our approach and conducted two case studies. In the first case-study, a controller for a motor, all errors could be found automatically within two hours. This shows that model-based testing for a relatively small system is profitable. The second case-study was done on a larger system and three errors were found. These errors were not found by manual testing. This shows that model-based testing in industrial automation is useful and saves costs and time. It is also more thorough than manual testing.

Future work will focus on incorporating real-time aspects in this setting using the theory in [24].

Acknowledgement. The authors wish to thank Prof. Dr. Ir. Joost-Pieter Katoen for comments on an earlier version of this paper.

References

1. Albrecht, H.: On Meta-Modeling for Communication in Operational Process Control Engineering VDI-Verlag (2003), VDI Fortschritt-Bericht, Series 8, No. 975, Düsseldorf, Germany, 3-18-397508-4, RWTH Aachen University
2. Bauer, N., Engell, S., Huuck, R., Lohmann, S., Lukoschus, B., Remelhe, M., Stursberg, O.: Verification of PLC Programs Given as Sequential Function Charts. In: Ehrig, H., Damm, W., Desel, J., Große-Rhode, M., Reif, W., Schnieder, E., Westkämper, E. (eds.) INT 2004. LNCS, vol. 3147, pp. 517–540. Springer, Heidelberg (2004)
3. Bauer, N., Huuck, R., Lukoschus, B., Engell, S.: A Unifying Semantics for Sequential Function Charts. In: Ehrig, H., Damm, W., Desel, J., Große-Rhode, M., Reif, W., Schnieder, E., Westkämper, E. (eds.) INT 2004. LNCS, vol. 3147, pp. 400–418. Springer, Heidelberg (2004)
4. Bornot, S., Huuck, R., Lukoschus, B., Lakhnech, Y.: Verification of Sequential Function Charts Using SMV. In: PDPTA 2000: International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, pp. 2987–2993 (2000)

5. Belinfante, A.: JTorX: A Tool for On-Line Model-Driven Test Derivation and Execution. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 266–270. Springer, Heidelberg (2010)
6. Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., Pretschner, A. (eds.): Model-Based Testing of Reactive Systems. LNCS, vol. 3472. Springer, Heidelberg (2005)
7. Frantzen, L., Tretmans, J., Willemse, T.A.C.: Test Generation Based on Symbolic Specifications. In: Grabowski, Nielsen (eds.) [19], pp. 1–15
8. Frantzen, L., Tretmans, J., Willemse, T.A.C.: A Symbolic Framework for Model-Based Testing. In: Havelund, K., Núñez, M., Roşu, G., Wolff, B. (eds.) FATES/RV 2006. LNCS, vol. 4262, pp. 40–54. Springer, Heidelberg (2006)
9. Grabowski, J., Nielsen, B. (eds.): FATES 2004. LNCS, vol. 3395. Springer, Heidelberg (2005)
10. Hartman, A., Nagin, K.: The AGEDIS Tools for Model Based Testing. SIGSOFT Softw. Eng. Notes 29(4), 129–132 (2004)
11. Hellgren, A., Fabian, M., Lennartson, B.: On the Execution of Sequential Function Charts. Control Engineering Practice 13, 1283–1293 (2004)
12. IEC International Electrotechnical Commission. IEC60848: GRAFCET Specification Language for Sequential Function Charts (2002)
13. IEC International Electrotechnical Commission. IEC 61131-03: Programmable Controllers - Part 3: Programming Languages, 2nd edn. (2003)
14. IEC International Electrotechnical Commission. IEC61512-2: Batch Control - Part 2: Data Structures and Guidelines for Language (2001)
15. IEC International Electrotechnical Commission. IEC62541-5: OPC Unified Architecture (2001)
16. Iyengar, P., Pulvermueller, E., Westerkamp, C.: Towards Model-Based Test Automation for Embedded Systems using UML and UTP. In: ETFA 2011. IEEE (2011)
17. Kumar, B., Czybik, B., Jasperneite, J.: Model Based TTCN-3 Testing of Industrial Automation Systems - First results. In: ETFA 2011. IEEE (2011)
18. Jard, C., Jéron, T.: TGV: Theory, Principles and Algorithms: A Tool for the Automatic Synthesis of Conformance Test Cases for Non-Deterministic Reactive Systems. J. STTS 7(4), 297–315 (2005)
19. McMillan, K.L.: Symbolic Model Checking. Kluwer Academic Publishers (1993)
20. Meyer, D.: Objektverwaltungskonzept für die operative Prozessleittechnik. VDI-Verlag (2002), VDI Fortschritt-Bericht, Series 8, No. 940, Düsseldorf, Germany, 3-18-394008-6, RWTH Aachen University
21. PLCopen. Technical Committee 6 Technical Paper: XML Formats for IEC 61131-3. Version 2.01 - Official Release (2009)
22. STSSimulator homepage, <http://java.net/projects/stssimulator/>
23. von Styp, S., Yu, L.: Two Case Studies for Applying Model Based Testing in Industrial Automation, AIB-2013-11, RWTH Aachen (2013)
24. von Styp, S., Bohnenkamp, H., Schmaltz, J.: A Conformance Testing Relation for Symbolic Timed Automata. In: Chatterjee, K., Henzinger, T.A. (eds.) FORMATS 2010. LNCS, vol. 6246, pp. 243–255. Springer, Heidelberg (2010)
25. von Styp, S., Yu, L., Quiros, G.: Automatic Test-Case Derivation and Execution in Industrial Control. In: iATPA 2011: First Workshop on Industrial Automation Tool Integration for Engineering Project Automation. CEUR-WS, pp. 7–12 (2011)

26. Tretmans, J.: Test Generation with Inputs, Outputs and Repetitive quiescence. *Software - Concepts and Tools* 17(3), 103–120 (1996)
27. UPPAAL homepage, <http://www.uppaal.org>
28. VDI/VDE Society for Measurement and Automatic Control. VDI/VDE 3681 Guideline: Classification and Evaluation of Description Methods in Automation and Control Technology (2005)
29. Yu, L., Quirós, G., Grüner, S., Epple, U.: SFC-Based Process Description for Complex Automation Functionalities. In: *EKA 2012: Entwurf Komplexer Automatisierungssysteme*, 12. Fachtagung, pp. 13–20. ifak, Magdeburg, Germany (2012)