Automatic Test-Case Derivation and Execution in Industrial Control

Sabrina von Styp
RWTH Aachen University
Chair of Software Modelling and Verification
Ahornstrasse 55, 52074 Aachen, Germany
von-styp@cs.rwth-aachen.de

Gustavo Quirós, Liyong Yu
RWTH Aachen University
Chair of Process Control Engineering
Turmstrasse 46, 52064 Aachen, Germany
{g.quiros, l.yu}@plt.rwth-aachen.de

Abstract

In industrial control, system failures can be highly dangerous and expensive. To avoid them, much time and money is spent in testing the systems, and this occurs mostly manually. Model-based testing is a forthcoming and promising technique to automate the testing process, as it allows the automatic generation and execution of test-cases. It assumes the system under test to be a black-box, implying that the system is only accessible via inputs and outputs. Thus, model-based testing can be ideally applied to function blocks. This paper presents our first steps in applying model-based testing in industrial control, and includes a case study on a simple motor controller.

1. Introduction

Controllers used in industrial settings often control highly critical systems, and their failure can be expensive due to loss of production and system damage. Therefore, testing a controller is crucial before employing it in practise. Currently this is usually done manually, and hence consumes a considerable amount of time and money.

One promising technique to automate the test process is model-based testing. In model-based testing, the test-cases are automatically driven from a formal model, the specification. The execution of these test-cases is commonly far more efficient and often of better quality than manually-crafted test-cases. Contrary to simulation, model-based testing requires no complex computation environment and the test-cases are derived from the specification, independently of the underlying physical model of the controlled plant. The advantages of this approach are the direct interplay between the real implementation and the specification – which allows for direct testing of specified behaviours, and the needlessness of calculating physical parameters. Another advantage of model-based testing is the higher test coverage. This is due to the systematic execution of test-cases independently of their likelihood in a real physical environment. Therefore, this approach is of high value for quality assurance of critical software.

In model-based testing, the system under test is considered to be a black-box, meaning that it can only be accessed and observed through its interfaces. A well-known representative of this technique is the $ioco$ framework [16]. It defines formal correctness criteria allowing the automatic generation of test-cases from the specification, which is a formal model given as a labelled transition system. It can be shown that the execution of such a derived test-case does not yield false positives, such as test failures, even though the implementation is correct.

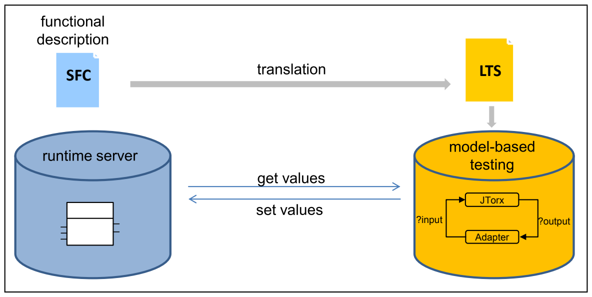

Figure 1. Overview

In this paper the authors show how model-based testing can be applied in industrial control, and present how errors can be detected quickly. Figure 1 provides an overview of our model-based testing approach. The starting point is the specification, which is given as a Sequential Function Chart (SFC), and the implementation running on a runtime server. First, the SFC is translated into a Labelled Transition System (LTS), which is then used by the test tool to derive and execute the test-cases. The test tool then executes the test by communicating with the implementation. The paper is structured as follows. The following section gives an introduction to model-based testing and its underlying theory. Afterwards, SFC as a specification language is introduced. Section 4 shortly presents the ACPLT technologies, including the function block system iFBSpro. In Section 5 the approach is explained, including a translation from SFC to LTS, which is needed for our model-based testing. The penultimate section presents our case-study on a simple motor controller. Finally the authors conclude this paper and give an insight on ongoing and future work.

2. Model-Based Testing

Model-based testing is a widely used technique for automating the test process. Test-cases are automatically driven from a formal model specifying the system under test. These test-cases are then executed against the real implementation. The Implementation Under Test (IUT) is considered to be a black-box, and therefore its correctness can only be verified by checking the response of the system for given stimuli. In order to define a formal correctness criteria it is assumed that the IUT can be represented as a formal model. Based on this assumption, referred to as test hypothesis, a testing theory is defined, including an implementation relation and an algorithm for automatic test-case derivation.

One well-known testing theory is the $ioco$ framework [16, 17] where the correct behaviour of an IUT is specified using an LTS with input and output actions. Test-cases cast their verdicts following a formal correctness criterion: the implementation relation $ioco$. The $ioco$ relation expresses that an implementation may only produce outputs if those outputs are also produced by the specification.

There exist several tools for the derivation of $ioco$ test-cases, e.g. TorX [5], TGV [8] and AGEDIS TOOL SET [1]. Based on the $ioco$-framework there is the $tioco$-framework [12] for real-time behaviour, the $sioco$-framework [11] for using data and the $stioco$-framework [18] for real-time behaviour including data.

2.1 Model-based testing with JTorX

The test-tool JTorX [9] is a platform-independent tool for model-based testing. It automatically derives test-cases from a given specification using the $ioco$ relation. The specification can be provided in the Aldebaran format .aut or in GraphML. In order to communicate with the implementation 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.

3. SFC for Functional Description

In today's automation environment, it is difficult for users to accept a control system that is incompatible with the IEC 61131-3 standard [7]. SFC is the only language in the IEC 61131-3 for discrete processes, and its graphical presentation makes the description of functionalities simple, intuitive and easily understandable.

SFC follows the specification language $GRAFCET$ [6], which is based on Petri nets. As a powerful programming language for the implementation of automation functions, SFC is also suitable for function descriptions. Sequential functions of any complexity can be precisely described by simple graphical elements: steps and transitions between steps. All SFC should 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 processes, which only need to be worked through once. In turn, the later one can describe a permanently active state machine.

SFC makes the unification of languages for description and implementation possible. The designer can specify the required sequential function by drawing SFC. Then, according to the used hardware and the specific running conditions, the SFC specification can be transformed to a directly executable SFC implementation. This unified description language can greatly shorten the development cycle of automation functionalities.

There are also cases of some execution 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 the requirements given by the designer with the help of the SFC graphic; on the other hand, if an existing implementation needs to be optimized, its SFC description can help the engineer to understand the working principle and to define solutions. In this paper we show how SFC descriptions can also be used for automatic testing by deriving test-cases from them and automatically executing these test-cases.

4. ACPLT Technologies

The ACPLT technologies [2] are reference models and software implementations that target various application areas within the field of process control engineering. This section describes those ACPLT technologies which are part of our model-based testing approach.

ACPLT/KS [3] is an open-source client/server communication system 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/OV [13] is an open-source object management and runtime environment, which permits the development of object-oriented applications that can operate in real-time process control environments. An ACPLT/OV application is hosted in a server which contains an object model, and the server offers an ACPLT/KS interface which permits remote clients to interact with the object model contained in the server.

The iFBSpro function block system [14] is a commercial engineering environment for developing applications using the model of Function Block Diagrams (FBD) [7], and is based on ACPLT/KS and ACPLT/OV. iFBSpro clients can communicate with iFBSpro servers with the purpose of performing engineering and monitoring tasks.

5. Approach

In order to use model-based testing, a formal description for the input and output behaviour in form of a transition system is needed. Therefore, the first step is to translate the given SFC into an LTS. Afterwards, an adapter is generated to enable the communication between the test-tool JTorX and the iFBSpro runtime server.

5.1 SFC to LTS

For Model-based testing the parameters of interest of an SFC are those that can be observed or changed from outside. Both [10] and [4] present a semantic for SFC using timed automata (TA). The aim of these two papers is to provide a formalism allowing formal verification of the model, i.e. model-checking. The resulting models consider more than just the observable input and output actions and are therefore not suitable for model-based testing, for which a simpler model is sufficient.

In this paper only controllers with at least one incoming and one outgoing variable are considered. The vector $\mathcal{I}$ represents all incoming variables, set by external events, from now on referred to as the environment. Outgoing variables $\mathcal{O}$ are changed in every step by the controller itself and are observed by the environment. The main idea of obtaining an LTS from a given SFC is that the transitions of the SFC are inputs of the LTS, and that the action steps of the SFC are outputs of the LTS. This means that every input transition of the LTS consists of an vector $\mathcal{I}$ containing the values of the input variables, and the output transition contains the output vector $\mathcal{O}$ accordingly. Steps are executed once per cycle and are repeated until at least one of its following transitions is enabled. The enabledness of a transition is checked at the end of each cycle.

Figure 2 shows a simple SFC. After starting, transition $T01$ is checked and in case it is true, `Step01` is executed for as long as at least either $T02$ or $T03$ is true. If both transitions are true, then $T02$ is executed since its priority is higher. Depending on the previous choice either `Step02` or `Step03` is executed. Figure 3 is the corresponding LTS representing the input and output behaviour of the SFC. The input actions, marked with a question mark, represent all concrete valuations of an input-vector that satisfy the corresponding transition, e.g.

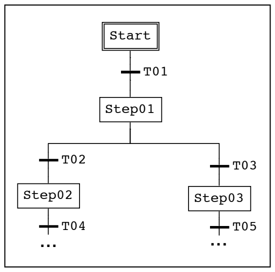

Figure 2. Example SFC

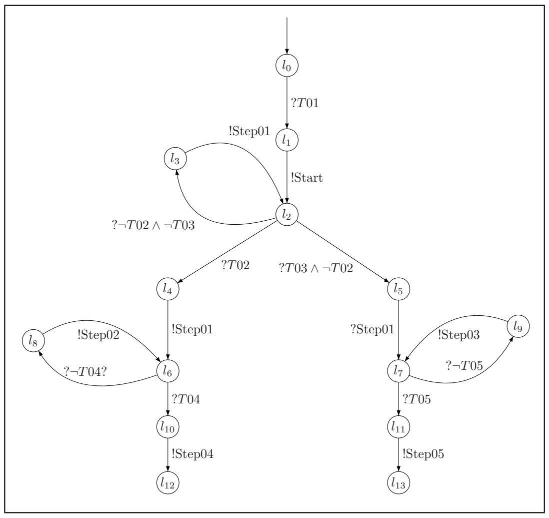

Figure 3. Example LTS

$?\neg T02 \land \neg T03$ stands for the set of input-vector valuations which satisfy neither $T02$ nor $T03$. The output actions, marked with an exclamation mark, represent the output-vectors that are observed after execution of a step. In Figure 3 the first transition is the input $?T01$, enabling the first transition $T01$. In location l_1 the systems then runs for exactly one cycle and executes the action `Start`. From location l_2 three different input actions can happen. These inputs set the variables of the transitions and are set while the system under test is offline. The action $?T02$ represents all possible variable settings that satisfy the conditions in $T02$. Note that there can be more than one transition from location l_2 to l_4, since there can be more than one variable valuation that satisfies $T02$, and LTS only allow concrete values and no variables. In Figure 2 transition $T03$ is only taken if $T02$ is not enabled. This is translated into the input $?T03 \land \neg T02$. If neither $T02$ nor $T03$ is enabled, represented by the input $?\neg T02 \land \neg T03$, `Step01` is executed and the system returns to location l_2. If input $?T02$ is taken, the system executes `Step01` and waits afterwards for transition $T04$ to hold. It cannot be guaranteed that the values for a transition, if set during a cycle, are set before the enabledness is checked. Hence, the values for the transitions are always set exactly between two cycles, before executing the actual step. E.g. in order to execute `Step01` in Figure 2 exactly once either $T02$ or $T03$ have to be set when the system is at `Step01` but has not executed the step yet. If neither $T02$ nor $T03$ are enabled, the system repeats Step01. This is represented by the loop in location l_3 in Figure 3.

5.2 Communication between Test-Tool and Implementation

The name and the format of the inputs and outputs that are provided and observed – respectively – by JTorX, are given by the specification and therefore do not necessarily conform with those that are used by the implementation that is running in the runtime server. Also, JTorX does not have a direct way of setting or getting variable values of the tested system, but the iFBSpro 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. The first problem can be solved by providing a file that maps the names and formats of the used variables. The second and third problems require an adaptor that sends and receives variable values in between JTorX and the server, and which also performs the required synchronisation by starting and stopping the implementation running on the server.

The adaptor is written in Java and communicates via standard input and output with JTorX while interacting with the server through ACPLT/KS. Paths of the variable locations on the iFBSpro server and it corresponding names in JTorX are provided by a text file.

During testing, the implementation is initially online but deactivated. Since the system has to obtain an initial input in order to start, the adaptor first waits for a stimulus from JTorX. This input is a string containing all variable settings and is parsed by the adaptor to set the values of the corresponding variables at the online server. Afterwards, a control variable is set so that the implementation runs exactly for one cycle. When the program has stopped the outputs are read and sent to JTorX which again sends the new stimuli. The adaptor itself runs until it receives a quit signal from JTorX. This can happen either when the test-execution from JTorX is stopped manually or if JTorX observes an incorrect output from the implementation.

6. Case Study

In order to evaluate the practical feasibility of our approach, an actual function block was specified, implemented and tested. The results of this evaluation are presented in this section.

6.1 The `simpleMotor` Function Block

For our case study, we chose a simple function block that controls an on/off motor. The function block has the following inputs, which are all of Boolean type: Con indicates that the motor should be switched on; Coff indicates that the motor should be switched off; 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 of the function block 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 may be 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 function block described above has been specified by means of the *SFC* that is depicted in Figure 4. 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. Both actions and transitions were formulated using the *FBD* language [7].

Based on the *SFC* specification, the simpleMotor function block was implemented in iFBSpro using the C language. In addition to this object class, five test classes named simpleMotorTest1, . . . , simpleMotorTest5 were produced as exact copies of the original class, but with manually introduced errors, and with the purpose of validating the model-based testing approach.

6.2 Test Execution

The description of the behaviour of the simpleMotor implementation is given as an *SFC*. Since JTorX requires this to be given as an *LTS*, the first step is to derive the *LTS* describing the input/output behaviour, from the *SFC* description as presented in Section 5.1. For this case study the derivation was done manually, but the authors are working one an automatic translation from PLCopen XML, an XML format for IEC 61131-3 languages [15], to the Aldebaran format .aut, which is also based on XML. The transition system generated from the simpleMotor specification has 23 locations and 33 transitions. It is shown in Figure 5. Here one can see that always the complete output vector is observed. Expres-

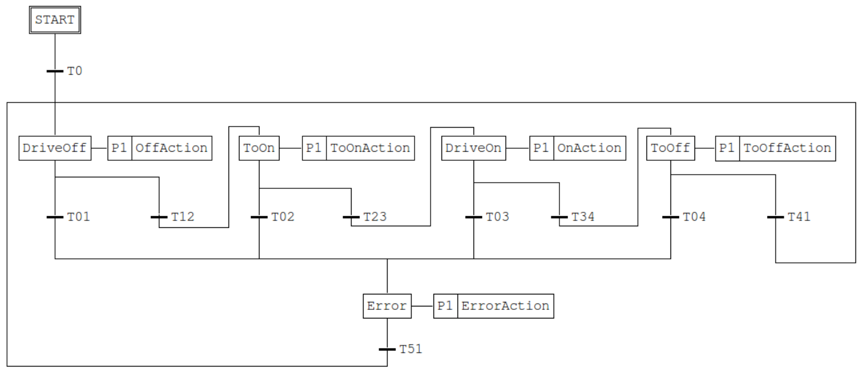

Figure 4. SFC specification for the `simpleMotor` function block

sions like *DriveOn-0* are the concrete labels standing for the value of the corresponding variable like *DriveOn* being false. In the first phase the generated transition system and the `simpleMotor` function block were used. The system under test was executed for 100.000 steps which took 190 minutes. Most of this time is due to the graphical output of JTorX and it can be reduced to 15 minutes if executed without the graphical output. The testing was stopped manually. This means that since inputs and outputs alternate, the implementation received 50.000 inputs and the same amount of outputs were observed and validated by JTorX. The test was stopped manually as it can be assumed that due to the state space of the transition system, there are practically no uncovered errors left.

In the second phase of the case-study the five versions named `simpleMotorTest1`, ..., `simpleMotorTest5` of the implementation were given. They were tested without knowing which version had been manipulated. All errors were found within an hour. There was no false negative.

JTorX produces three different kinds of graphical outputs. The first one is the transition system of the specification highlighting were JTorX stopped in case of an error. The second one shows the traces that have been executed, and the last one is a message sequence chart. In our case study, the implementation is first started with input *?true*, leading to location 18 in Figure 5. The output *!DriveOn-0_Act-0_DriveOff-0_Err-0* is then observed. The next two inputs enable neither transition `T01` nor `T12` and the controller loops in the `DriveOff` state, represented with the loop of location 17 and 19 in Figure 5. The fourth input *?chkb_On-1* sets variable chkb_On to true and, after executing step `DriveOff` one last time, leads to the `Error` step and location 3 in our LTS. After the input *?C_Ack-1* the output *!DriveOn-0_Act-0_DriveOff-1_Err-1* is observed but as shown in the LTS, output *!DriveOn-0_Act-0_DriveOff-0_Err-1* is expected by JTorX. This then leads to the termination of the test producing the verdict *fail*.

```
case Error :
  /* ErrorAction */
  v_ACT = FALSE; v_DriveOn = FALSE; v_ERR = TRUE;
  /* T51 */
  transition (v_CACK == TRUE, DriveOff);
```

The code above shows a part of the program code and the error that was manually created. Every variable is set correctly except for `DriveOff`, whose value is not set at all and therefore leads to the error detected above.

In this case study the system under test only has P1 action qualifier. It also works for P0 and N with the restriction that variables are not allowed to be changed in every loop, i.e. such as incrementing or decrementing a variable. Since LTS work with concrete transition, changing the variables in every loop leads to state space explosion and therefore is not applicable in practice.

7. Perspective

This paper demonstrated an approach to apply automatic test-case generation and execution in industrial control. It gave an introduction on model-based testing, SFC as a specification language and ACPLT technologies, and afterwards presented a translation from SFC to LTS, which is needed in order to apply model-based testing by using the tool JTorX. The case-study presented proved the applicability of this approach to real implementations. The execution of the generated test-cases was both efficient in execution time and effective in finding all errors in the implementation. Nevertheless, the motor controller used in the case-study is a small and restricted implementation. It has only Boolean variables, and has no timing constraints. The problem with using variables other than Booleans is the state space explosion that would occur if the transition system of such an SFC was obtained. A constraint like 0<x>10 in a transition of an SFC, where x is a double precision floating point number with two digits, already leads to 1000 transitions in the corresponding LTS. In order to avoid such a state space explosion, future research will be to use the same concepts as presented in this paper but using symbolic transition systems rather then labelled transition systems. These transition systems allow a notion of data and data dependent control flow based on first order logic formulas. Finally, the authors also plan to extend our approach to allow timing

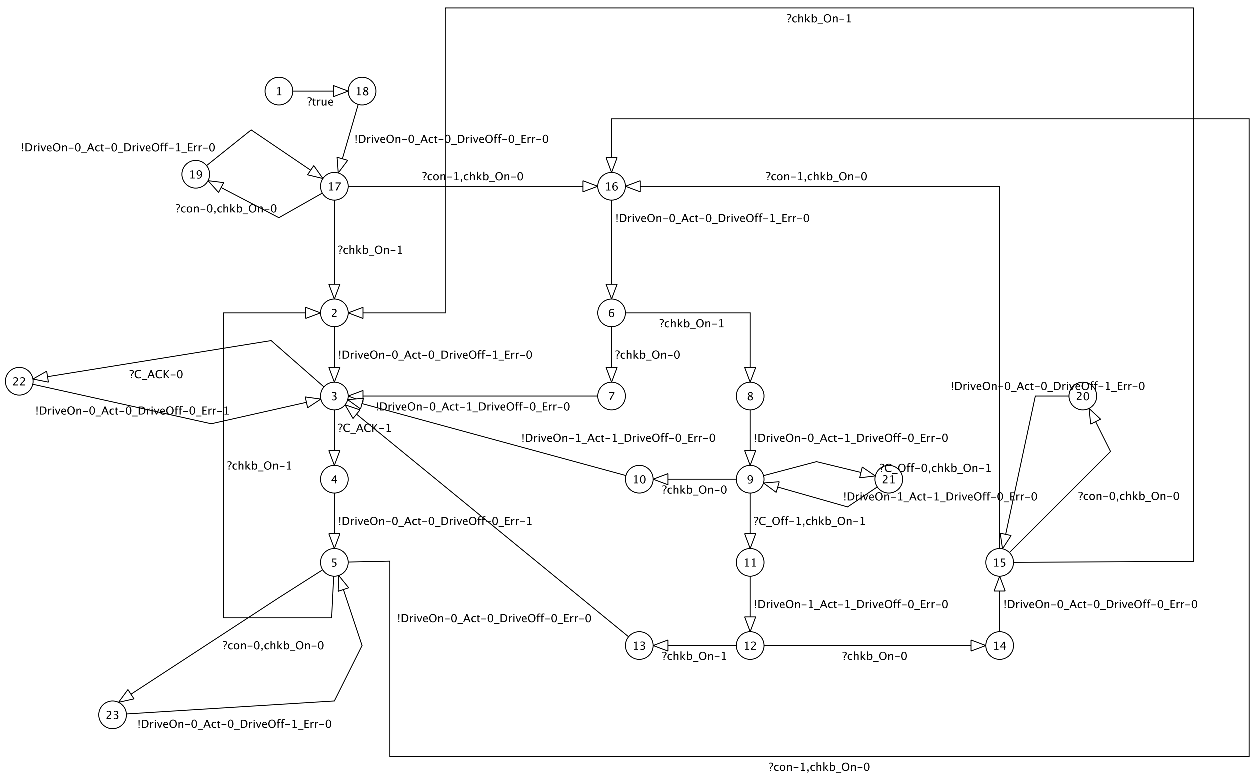
?chkb_On-1
1
?true
18
!DriveOn-0_Act-0_DriveOff-1_Err-0
!DriveOn-0_Act-0_DriveOff-0_Err-0
19
17
?con-1,chkb_On-0
16
?con-1,chkb_On-0
?con-0,chkb_On-0
!DriveOn-0_Act-0_DriveOff-1_Err-0
?chkb_On-1
2
6
?chkb_On-1
!DriveOn-0_Act-0_DriveOff-1_Err-0
?chkb_On-0
?C_ACK-0
22
3
7
8
!DriveOn-0_Act-0_DriveOff-1_Err-0
20
!DriveOn-0_Act-1_DriveOff-0_Err-0
!DriveOn-0_Act-0_DriveOff-0_Err-1
?C_ACK-1
!DriveOn-1_Act-1_DriveOff-0_Err-0
!DriveOn-0_Act-1_DriveOff-0_Err-0
?chkb_On-1
4
10
9
?C_Off-0,chkb_On-1
21
?chkb_On-0
!DriveOn-1_Act-1_DriveOff-0_Err-0
?con-0,chkb_On-0
!DriveOn-0_Act-0_DriveOff-0_Err-1
?C_Off-1,chkb_On-1
5
11
15
!DriveOn-1_Act-1_DriveOff-0_Err-0
!DriveOn-0_Act-0_DriveOff-0_Err-0
!DriveOn-0_Act-0_DriveOff-0_Err-0
?con-0,chkb_On-0
13
12
14
?chkb_On-1
?chkb_On-0
23
!DriveOn-0_Act-0_DriveOff-1_Err-0
?con-1,chkb_On-0

Figure 5. simpleMotor LTS

constraints.

References

[1] K. N. A. Hartman. The AGEDIS tools for model based testing. *SIGSOFT Softw. Eng. Notes*, 29(4):129–132, 2004.

[2] ACPLT. http://www.acplt.org.

[3] H. Albrecht. *On Meta-Modeling for Communication in Operational Process Control Engineering*. VDI Fortschritt-Bericht, Series 8, No. 975. VDI-Verlag, Düsseldorf, Germany, 2003.

[4] N. Bauer, S. Engell, R. Huuck, B. Lukoschus, and O. Stursberg. Verification of PLC programs given as Sequential Function Charts. In *Integration of Software Specification Techniques for Applications in Eng., Springer, LNCS*, volume 3147, pages 517–540, 2004.

[5] A. Belinfante. JTorX: A tool for on-line model-driven test derivation and execution. In J. Esparza and R. Majumdar, editors, *TACAS 2010*, volume 6015 of *LNCS*, pages 266–270. Springer-Verlag, 2010.

[6] IEC, International Electrotechnical Commission. *IEC 60848: GRAFCET specification language for sequential function charts*, 2002.

[7] IEC, International Electrotechnical Commission. *IEC 61131-3 Ed. 2.0: Programmable controllers — Part 3: Programming languages*, 2003.

[8] C. Jard and T. Jéron. 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.

[9] JTorX. http://fmt.cs.utwente.nl/redmine/wiki/jtorx/.

[10] S. Kowalewski, S. Engell, R. Huuck, Y. Lakhnech, B. Lukoschus, and L. Urbina. Using model-checking for timed automata to parameterize logic control programs. In *In Proc. of the 8th European Symp. on Computer Aided Process Engineering (ESCAPE*, pages 24–27, 1998.

[11] T. W. L. Frantzen, J. Tretmans. A Symbolic Framework for Model-Based Testing. In K. Havelund, M. Núñez, G. Rosu, and B. Wolff, editors, *FATES/RV 2006*, volume 4262 of *LNCS*, pages 40–54. Springer, 2006.

[12] S. T. M. Krichen. Black-box conformance testing for real-time systems. In *SPIN 2004*, volume 2989 of *LNCS*, pages 109–126. Springer, 2004.

[13] D. Meyer. *Objektverwaltungskonzept für die operative Prozessleittechnik*. VDI Fortschritt-Bericht, Series 8, No. 940. VDI-Verlag, Düsseldorf, Germany, 2002.

[14] J. Nagelmann and A. Neugebauer. *iFBSPro Version 2.0*. LTSoft GmbH, Kerpen, Germany, Feb. 2007.

[15] PLCopen. http://www.plcopen.org.

[16] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools*, 17(3):103–120, 1996.

[17] J. Tretmans. Model based testing with labelled transition systems. In *Formal Methods and Testing*, pages 1–38, 2008.

[18] S. von Styp, H. Bohnenkamp, and J. Schmaltz. A conformance testing relation for symbolic timed automata. In *Proc. FORMATS 2010*, volume 6246 of *LNCS*, pages 243–255. Springer-Verlag, 2010.