Test Automation for Safety-Critical Systems: Industrial Application and Future Developments

Jan Peleska

JP Software-Consulting and University of Bremen*

Abstract. Design, execution and evaluation of tests for safety-critical systems require considerable effort and skill and consume a large part of today's development costs. Due to the growing complexity of control systems, it has to be expected that their trustworthy test will become unmanageable in the future, if only conventional techniques, requiring a high degree of human interaction during the test process, are applied. In this article, we will focus on test automation for reactive real-time systems, with emphasis on Hardware-in-the-Loop tests analyzing the behaviour of combined software and hardware components. To illustrate possible approaches for this test problem, we describe a concept based on specifications written in Real-Time CSP. For the implementation of test generation and evaluation algorithms transition system representations are used, as can be obtained by Formal Systems' FDR tool. An industrial application of the method is presented and used for the evaluation of the benefits of formal methods-based testing in comparison with conventional techniques. Furthermore, we will indicate research topics in this field which are likely to become important for further improvements of the test process. Specifically, the benefits arising from an approach combining formal verification and testing will be discussed. Our presentation aims less at "promoting" a specific solution, but tries to illustrate the basic problems to be tackled with any formal method, when trying to develop test automation concepts to be applied in the context of reactive systems.

Keywords: CSP — FDR — reactive systems — refinement — test driver — test generation — test monitors — test oracles

1 Introduction

1.1 Objectives

The objective of this article is to outline some of the major problems to be tackled in the field of test automation. The growing demand for trustworthy, but at the same time manageable and cost effective test and verification methods on one hand and the growing number of publications and prototype tools focusing on

* Universität Bremen, Fachbereich 3, D-28334 Bremen, Germany, e-mail: jp@informatik.uni-bremen.de

to analyze the correctness of hardware/software cooperation. These considerations have resulted in ***hardware-in-the-loop tests*** being the most important testing technique for such systems. A hardware-in-the-loop test system consists of one or more computers connected to the target system by the interface designed for the "real" operational environment plus additional monitoring channels used to record or even manipulate internal states of the target system.

Let us look at the logical building blocks of a complete hardware-in-the-loop test automation system (Figure 1)[3].

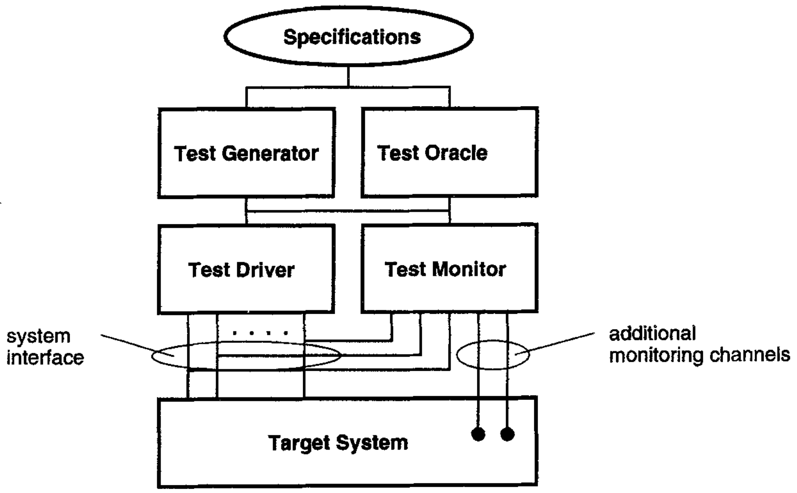

Fig. 1. Logical building blocks of a test automation system.

The Test Generator is responsible for the creation of test cases from specifications. The generator is called trustworthy, if for each possible implementation error violating the specified requirements a test case will be created which is capable of detecting this error. Observe that in the context of reactive systems, a test case is not simply a sequence of inputs and expected outputs: Since the target system may behave nondeterministically on the interface level, the same sequence of inputs may stimulate different responses of the target system, or may even be refused during repeated executions. Therefore we rather define a test case as a description of ***execution rules***, specifying the full set of possible sequences of input and output events, together with ***real-time assertions*** which should be met when exercising the test case on the target system.

The Test Driver interprets the test cases provided by the test generator and controls their execution by writing data on input channels of the target system and collecting system outputs. In general, the target system is only required to behave according to the specification, as long as the operational environment

[3] Figure 1 is not intended to suggest a system architecture for a test system, it only represents logical components that should be present.

also behaves correctly. A test driver is therefore called trustworthy, if it exactly simulates the behaviour of the "real" environment during test execution.

The **Test Oracle** evaluates the observed test execution against a specification and decides whether the execution was correct. The specification document used for test evaluation, say, $SPEC_2$ is not necessarily the same as the specification $SPEC_1$ used for test generation: For example, $SPEC_1$ may be an explicit CSP specification, because the possibility to interpret explicit CSP processes is suitable for test generation. For the test oracle, it may be preferable to use trace assertions in $SPEC_2$. Moreover, it will not always be the case that every behavioural property reflected by $SPEC_1$ will also be checked by $SPEC_2$. Conversely, it may be the case that $SPEC_1$ is an incomplete description of the required behaviour, which is just suitable for the generation of certain test cases, while $SPEC_2$ is a stronger specification, valid for the evaluation of other classes of test cases as well. For the test oracle trustworthiness means that, given the results of a test execution, it will be detect every violation of the specified requirements.

The **Test Monitor** observes each test execution in order to decide whether (1) a specific test case has been performed for all relevant executions that are possible for this case, when exercised on the target system, and (2) the full set of test cases executed so far suffices for the required level of test coverage. In many applications, these tasks of a test monitor cannot be performed based on the black-box observations of the system interface alone. Instead, additional channels must be created, providing internal state information about the target system for the monitor.

The fundamental capabilities of a trustworthy test system can be summarized by two aspects [7]:

- *Unbias:* Correct target system behaviour is never rejected. For this property it has to be ensured that (1) the test generator only creates tests which should really be successfully executed according to the specification, (2) the test driver simulates the expected environment behaviour in a proper way and (3) the test oracle recognizes the test execution to be consistent with the specification.
- *Validity:* Only correct target system behaviour is accepted. For this property it has to be ensured that (1) the test generator is capable to create tests that can uncover each possible correctness violation, (2) the test driver simulates the expected environment behaviour in a proper way, (3) the test oracle recognizes the test execution to be inconsistent with the specification and (4) the test monitor detects whether the test executions so far have covered all relevant target system behaviours.

3 A Tramway Crossing Control System

In this section a *Tramway Crossing Control System (TCCS)* is informally described. The TCCS has been developed by ELPRO LET GmbH in Berlin, Gerinserted between peripherals and Control Module that transform input raw data into abstracted values and refine the logical commands issued by the Control Module according to the requirements of the TCCC output interfaces. For example, the Track Monitor receives raw signals from the track sensors. These signals are de-bounced and monitored with respect to steadiness over a certain time interval. After that they are accepted to be valid and passed to the Control Module as abstract events *in*1 (a train has passed track sensor TS1-IN), ..., *out*2 (a train has passed track sensor TS2-OUT). The TL-Interface Handler receives commands *red* = *On*/*OFF* and *yellow* = *ON*/*OFF* from the Control Module. These commands are transformed and multiplexed into the corresponding switching commands for both traffic lights. Moreover, the TL-Interface Handler monitors feedback lines from each traffic light indicating their actual status. To increase the reliability of the system, each lamp of a traffic light is equipped with a spare filament. This is automatically activated by the interface handler if the first one fails. The SIG-Interface Handler operates analogously.

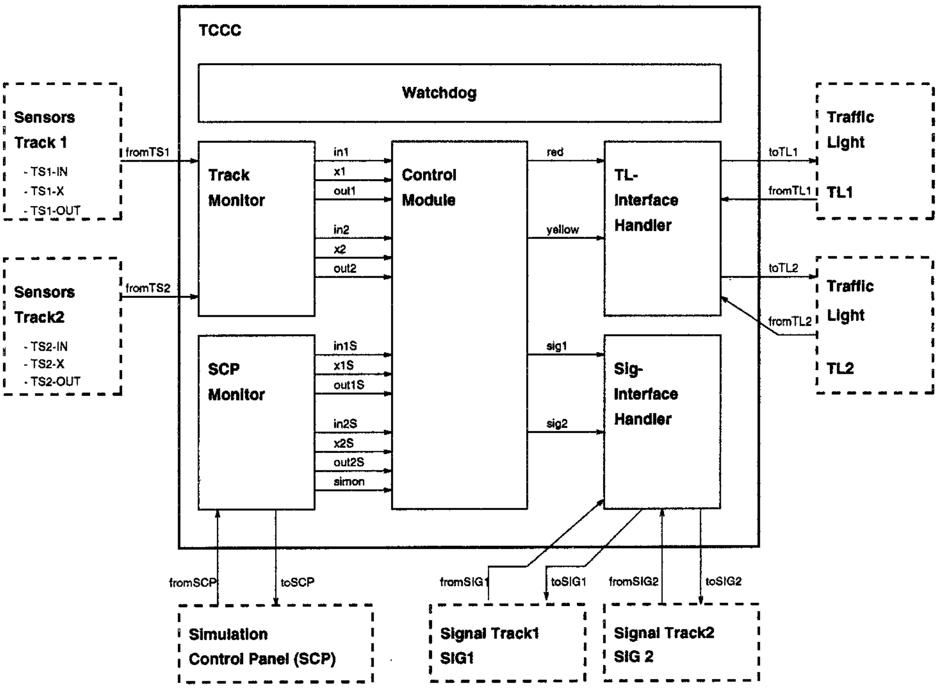

Fig. 3. Architecture of the Tramway Crossing Control Computer

The TCCC is controlled by a (hardware) Watchdog: if any of the interface or control modules detect a safety violation, the module directly triggers the watchdog to initiate a transition into a *stable safe state*, where the signals are automatically switched off and the traffic lights are switched to yellow-flashing.

This transition is completely performed by the watchdog hardware and does not require any cooperation of the software modules. The watchdog also detects hardware failures of the TCCC.

4 Test Automation for the TCCS

In this section we illustrate the concepts presented in Section 2 by describing the test automation process performed for the TCCS introduced in the previous section.

4.1 Functional Versus Structural Testing

For dynamic testing a basic distinction is made: *Functional* testing considers the target system as a black box and uses a specification of the required functionality *as far as visible at the system interface* for test generation, execution and evaluation[4]. *Structural* testing considers the system as a white box and derives test cases from the implemented structures, like branching of the code, the function call hierarchy, the data flow between processes etc. It is obvious to see that functional and structural testing are complementary: The former cannot provide sufficient test coverage because it does not consider internal design, the latter cannot detect missing functionality. It is less obvious whether more emphasis should be put on functional or structural testing.

For the test of the TCCS our decision was as follows: The main effort was invested in functional tests, with the objective to cover the specified system functionality as completely as possible. During functional testing, the internal system structures covered were recorded. Structural tests were only designed and executed to cover the remaining system structure not reached by the preceding functional tests.

This approach is motivated by the following facts: (1) The most severe design and implementation errors are caused by erroneous specifications or their incorrect interpretation. Therefore tests comparing the implementation behaviour to the specification are likely to detect the most critical errors. (2) While structural tests cannot be created before the completion of the design phase, functional tests can be derived as soon as the specification is ready. They can also be used to *validate* the specification and test the design before entering the implementation phase. (3) The specification contains all the requirements relevant for the user and – in case of dependability requirements for a safety-critical system – the certification authorities. Therefore the acceptance test was based on the specification. These considerations are consistent with the observations made in [20, 21, 11].

[4] To be more precise, the term *functional and behavioural testing* should be used, because in the context of reactive systems these tests have to analyze both the functionality (data transformations) and the behavioural properties (causal properties, synchronization and timing of events). However, the term *functional test* is so widely used that we do not wish to introduce a new term.

be expressed by the refinement relation

$$\begin{array}{l}
((CM \underset{\{\!|\, red, yellow \,|\!\}}{\parallel} (ATLIH \underset{\{\!|\, toTL1, fromTL1, toTL2, fromTL2 \,|\!\}}{\parallel} (TL1 \,|||\, TL2))) \\
\quad \underset{\{\!|\, sig1, sig2 \,|\!\}}{\parallel} (SIGIH \underset{\{\!|\, toSIG1, fromSIG1, toSIG2, fromSIG2 \,|\!\}}{\parallel} (SIG1 \,|||\, SIG2))) \\
\quad \setminus (\Sigma - \{\!|\, red, yellow, toTL1, fromTL1, toTL2, fromTL2 \,|\!\}) \\
\sqsubseteq_{TT} \\
((CM \underset{\{\!|\, red, yellow \,|\!\}}{\parallel} (TLIH \underset{\{\!|\, toTL1, fromTL1, toTL2, fromTL2 \,|\!\}}{\parallel} (TL1 \,|||\, TL2))) \\
\quad \underset{\{\!|\, sig1, sig2 \,|\!\}}{\parallel} (SIGIH \underset{\{\!|\, toSIG1, fromSIG1, toSIG2, fromSIG2 \,|\!\}}{\parallel} (SIG1 \,|||\, SIG2))) \\
\quad \setminus (\Sigma - \{\!|\, red, yellow, toTL1, fromTL1, toTL2, fromTL2 \,|\!\})
\end{array}$$

Application of the hiding operator indicates that for the tests or formal verification of *TLIH* against *ATLIH* events outside the unit interface were disregarded. Similarly, the verification of the Sig-Interface Handler unit is expressed as

$$\begin{array}{l}
((CM \underset{\{\!|\, red, yellow \,|\!\}}{\parallel} (TLIH \underset{\{\!|\, toTL1, fromTL1, toTL2, fromTL2 \,|\!\}}{\parallel} (TL1 \,|||\, TL2))) \\
\quad \underset{\{\!|\, sig1, sig2 \,|\!\}}{\parallel} (ASIGIH \underset{\{\!|\, toSIG1, fromSIG1, toSIG2, fromSIG2 \,|\!\}}{\parallel} (SIG1 \,|||\, SIG2))) \\
\quad \setminus (\Sigma - \{\!|\, sig1, sig2, toSIG1, fromSIG1, toSIG2, fromSIG2 \,|\!\}) \\
\sqsubseteq_{TT} \\
((CM \underset{\{\!|\, red, yellow \,|\!\}}{\parallel} (TLIH \underset{\{\!|\, toTL1, fromTL1, toTL2, fromTL2 \,|\!\}}{\parallel} (TL1 \,|||\, TL2))) \\
\quad \underset{\{\!|\, sig1, sig2 \,|\!\}}{\parallel} (SIGIH \underset{\{\!|\, toSIG1, fromSIG1, toSIG2, fromSIG2 \,|\!\}}{\parallel} (SIG1 \,|||\, SIG2))) \\
\quad \setminus (\Sigma - \{\!|\, sig1, sig2, toSIG1, fromSIG1, toSIG2, fromSIG2 \,|\!\})
\end{array}$$

Now a rather lengthy algebraic calculation shows that this implies the desired refinement property for the integrated units,

$$\begin{array}{l}
((CM \underset{\{\!|\, red, yellow \,|\!\}}{\parallel} (ATLIH \underset{\{\!|\, toTL1, fromTL1, toTL2, fromTL2 \,|\!\}}{\parallel} (TL1 \,|||\, TL2))) \\
\quad \underset{\{\!|\, sig1, sig2 \,|\!\}}{\parallel} (ASIGIH \underset{\{\!|\, toSIG1, fromSIG1, toSIG2, fromSIG2 \,|\!\}}{\parallel} (SIG1 \,|||\, SIG2))) \\
\sqsubseteq_{TT} \\
((CM \underset{\{\!|\, red, yellow \,|\!\}}{\parallel} (TLIH \underset{\{\!|\, toTL1, fromTL1, toTL2, fromTL2 \,|\!\}}{\parallel} (TL1 \,|||\, TL2))) \\
\quad \underset{\{\!|\, sig1, sig2 \,|\!\}}{\parallel} (SIGIH \underset{\{\!|\, toSIG1, fromSIG1, toSIG2, fromSIG2 \,|\!\}}{\parallel} (SIG1 \,|||\, SIG2)))
\end{array}$$

provided that (1) the alphabets of *TLIH* and *SIGIH* are disjoint and (2) both specifications and implementations of the interface handlers behave identically at their interfaces $\{\!|\, red, yellow \,|\!\}$ and $\{\!|\, sig1, sig2 \,|\!\}$ shared with the Control Module *CM*. The nature of these premises suggests not perform an integration test, but simply an inspection for their verification: (1) is implied by the fact that the implementations of the two interface handlers do not share any resources (variables, IO-channels), as may be verified by a static analyzer. (2) is implied by the fact that the Control Module only outputs to the interface handlers, and – due to the interface concept using shared variables – such an output is never refused. As a consequence *TLIH* and its specification *ATLIH* act like $RUN_{\{\!|\, red, yellow \,|\!\}}$ and *SIGIH*, *ASIGIH* act like $RUN_{\{\!|\, sig1, sig2 \,|\!\}}$ at the interface to *CM*.

Strategies Distinguishing Between Normal and Exceptional Behaviour A further important test strategy is to distinguish between *normal behaviour* and *exceptional behaviour* of the environment. It is often useful to start with tests only investigating the target system behaviour in presence of proper environment operation. The investigation whether the target system will react properly to abnormal environment behaviour is then tackled in a second test phase. This heuristic approach is also supported by theoretic considerations about dependable systems [19], so a test strategy separating normal and exceptional behaviour tests can be formally justified.

4.4 Preparing a Test Configuration

The application of various test strategies requires to test the target system in different environments, each one defined according to the specific restrictions of the actual testing stage. It is therefore desirable to generate each driver simulating a certain environment behaviour directly from specifications. Each test setup using a fixed environment is called a *test configuration*. We will now analyze the major steps required to construct such configurations.

Hardware-in-the-Loop Test Configuration To illustrate the architecture required for specific hardware-in-the-loop tests, let us suppose that we wish to set up a configuration for the test of the TCCC Control Module, without changing the TCCC structure depicted in Figure 3. Such a test configuration architecture is shown in Figure 4. The Environment Simulation Layer drives the test of the target component. The subordinate layers in the target system and the test system provide a *virtual data flow* between the two top-level layers. This will now be explained in more detail.

Defining Abstractions As pointed out in [11, 21], testing against specifications requires an *abstraction mapping* associating the concrete data and events generated by the target system with abstract objects in the specification. While Richardson [21] describes this task to be a "mapping of the name spaces", we think that more complex considerations are required in the context of reactive systems, because the abstraction is rather a *process* linking concrete and abstract layers. The mapping concept is more appropriate for tests involving data refinement. We will illustrate this observation using the TCCC test configuration introduced above.

For the purpose of testing the Control Module we think of the TCCC as divided into two layers: The Control Module represents the "visible" application layer, the other monitors and interface handlers are located in a subordinate layer supporting the Control Module communication.

The test driver part of the hardware-in-the-loop test system is subdivided into three layers: The Interface Driver Layer implements the low-level drivers needed to communicate raw data over the external interfaces $fromTR1, fromTR2, \ldots$ of the TCCC. These drivers have to be programmed (as far as not already available on sensor raw data and generate the corresponding $in1$-event for the Control Module, the raw data values on the two lines have to be steady for t_{sens} time units. After that they should be reset to their passive values $in1.high.0$ and $in1.low.1$ indicating that no train is passing the sensor.

In the other transmission direction, the Abstraction Layer abstracts received interface data by generating the corresponding events according to the inputs of the Environment Simulation Layer. Again, this has to be modelled by process specifications. To generate, for example, a $sig1'$-event at the Environment Simulation Layer in response to a $sig1$-event issued by the *Control Module*, the following process is required:

$$GenSIG1 = toSIG1?x \rightarrow fromSIG1!ack(x) \rightarrow sig'!x \rightarrow GenSIG1$$

On reception of an event $toSIG1.x, x = on, off$ the Abstraction Layer has to acknowledge this to the TCCC via $fromSIG1!ack(x)$ to simulate the correct operation of $SIG1$, otherwise the TCCC Watchdog would enforce a transition into the stable safe state. After having acknowledged the TCCC command, the associated abstract event $sig'!x$ is passed to the Environment Simulation Layer.

In the VVT-RT test system, process specifications are directly interpreted in real time, so that no programming effort is necessary to set up the Abstraction Layer.

Observe that the abstraction scheme has to be justified for each specific application, because the additional layers introduce a buffer and a time delay between the target component and the test driver. In case of the TCCC, the time delay and the additional level of buffering is unimportant, because the signal changes to be expected for each interface in the operational environment are much slower than the time needed to pass messages through the test system layers and the TCCC never refuses inputs. Furthermore note that the abstraction scheme described only holds for *deterministic* subordinate target system layers, as it is the case for the TCCC monitors and interface handlers. In case of nondeterministic behaviour, it would be uncertain whether an external stimulus (e. g., inputing a TS1-IN sensor signal at the $fromTS1$-interface with a nondeterministic Track Monitor) would really lead to the intended input at the target component interface (e. g., sending an $in1$-event to the Control Module). For these situations, additional monitoring channels have to be installed between the target system and the test system, so that the inputs really received and generated by the target component can be observed and used for test evaluation. In case of a high degree of nondeterminism it can be advisable to block the subordinate layer completely and use auxiliary channels to manipulate the inputs of the target system directly.

Specifying the Environment For the purpose of generating and driving tests we need a specification of the operational environment behaviour, while a specification of the target system is completely unnecessary for these activities: Any behaviour visible at the system interface and possible for the environment can be used for testing purposes, regardless of the intended behaviour of the target system. Even if the complete target system is required to operate in an arbitrary environment, environment specifications will be used to enforce the test strategies discussed above.

For the test of the TCCC using VVT-RT, the environment specification is an explicit timed CSP specification E, using untimed CSP syntax as accepted by Formal System's FDR tool [6] or a slightly restricted timed CSP syntax (see [16]) which is suitable for symbolic execution in real time. To allow test generation and symbolic execution, E is first transformed into a set of transition graphs [17] corresponding to each sequential process component of E. This transformation is performed by means of FDR. Real-time specification are pre-compiled into untimed specifications enriched by auxiliary timer events marking every "real" event associated with a time-depending refusal. The decomposition of E into several sequential components is performed for two reasons: First, to reduce the complexity of the transition graphs involved and facilitate their interpretation in real time. Second, to provide a basis for correct real-time simulation: In real-time specifications, the well-known semantic equivalence between distributed and nondeterministic sequential programs (see [1]) is no longer valid. As a consequence, a complete transition graph of a timed distributed program cannot be faithfully interpreted by one sequential test driver process. Instead, the test driver has to be implemented by parallel processes, each one interpreting the transition graph of a sequential environment component and cooperating with a scheduler observing the synchronization conditions between the sequential processes.

The transition graph representation of E will be executed in the Environment Simulation Layer. Note that if a configuration has to test the full target system, this layer directly receives and generates events of the system interface. In this case the Abstraction Layer is just the identity mapping, and only the correspondence of CSP events to interface driver calls is evaluated.

Let us consider the specification of normal environment behaviour, as it is used for the Control Module test configuration discussed above. A train number *trainNo* passing on track *trackNo* can be modelled as

$$\begin{aligned}
&\mathit{TRAIN}(\mathit{trackNo},\mathit{trainNo}) = \mathit{WAIT}\ 0\,..\,t_0;\ \mathit{inE}!\mathit{trackNo}.\mathit{trainNo}\\
&\quad\rightarrow \mathit{WAIT}\ t_1\,..\,t_2;\ \mathit{xE}!\mathit{trackNo}.\mathit{trainNo}\\
&\quad\rightarrow \mathit{WAIT}\ t_3\,..\,t_4;\ \mathit{outE}!\mathit{trackNo}.\mathit{trainNo} \rightarrow \mathit{TRAIN}(\mathit{trackNo},\mathit{trainNo})
\end{aligned}$$

$\mathit{WAIT}0\,..\,t_0$ denotes the nondeterministic choice of waiting for $0, 1, 2, \ldots$ or t_0 time units before entering the track near the crossing. $\mathit{inE}, \mathit{xE}, \mathit{outE}$ are auxiliary channels defined to control the generation of the track sensor events $\mathit{in}1, \mathit{in}2, \ldots$, as will be explained in the next specification. $\mathit{WAIT}\ t_1\,..\,t_2$ denotes minimal, maximal and further durations to be tested for the time required by the train to drive from the IN-sensor to the X-sensor. Analogously, $\mathit{WAIT}\ t_3\,..\,t_4$ denotes the durations to be tested for passing the track section starting with the X-sensor and ending with the OUT-sensor.

In normal operation, up to $\mathit{maxTrains} = 4$ trains may be located between the IN- and OUT-sensors. Trains will only pass an X-sensor if the corresponding signal is ON. Of course, trains cannot overtake each other. These conditions are modelled by[8]

$$
\begin{aligned}
&\mathit{Ectrl}1(\mathit{inq}, \mathit{xq}, \mathit{sigState}) = \\
&\quad (\#\mathit{inq} < \mathit{maxTrains})\&\mathit{inE}.1?\mathit{trainNo} \rightarrow \mathit{in}1' \\
&\qquad\qquad \rightarrow \mathit{Ectrl}1(\mathit{inq}^\frown\langle \mathit{trainNo}\rangle, \mathit{xq}, \mathit{sigState}) \\
&\quad \Box(\mathit{sigState} = \mathit{ON} \wedge 0 < \#\mathit{inq})\&\mathit{xE}.1!\mathit{head}(\mathit{inq}) \rightarrow x1' \\
&\qquad\qquad \rightarrow E1\mathit{ctrl}(\mathit{tail}(\mathit{inq}), \mathit{xq}^\frown\langle \mathit{head}(\mathit{inq})\rangle, \mathit{sigState}) \\
&\quad \Box(0 < \#\mathit{xq})\&\mathit{outE}.1!\mathit{head}(\mathit{xq}) \rightarrow \mathit{out}1' \\
&\qquad\qquad \rightarrow E1\mathit{ctrl}(\mathit{inq}, \mathit{tail}(\mathit{xq}), \mathit{sigState}) \\
&\quad \Box\, \mathit{sig}1'?z \rightarrow \mathit{Ectrl}(\mathit{inq}, \mathit{xq}, z)
\end{aligned}
$$

Each train generating an auxiliary inE-, xE- $outE$-event stimulates a corresponding $in1'$-, $x1'$-, $out1'$-event which is passed via abstraction layer and subordinate TCCC layers to the Control Module. $Ectrl2$ is specified accordingly. Finally, consecutive trains cannot pass the same sensor arbitrarily close:

$$\mathit{INsensor}(\mathit{trackNo}) = \mathit{inE}.\mathit{trackNo}?\mathit{trainNo} \xrightarrow{t_1} \mathit{INsensor}(\mathit{trackNo})$$

$$\mathit{Xsensor}(\mathit{trackNo}) = \mathit{xE}.\mathit{trackNo}?\mathit{trainNo} \xrightarrow{t_1} \mathit{Xsensor}(\mathit{trackNo})$$

$$\mathit{OUTsensor}(\mathit{trackNo}) = \mathit{outE}.\mathit{trackNo}?\mathit{trainNo} \xrightarrow{t_1} \mathit{OUTsensor}(\mathit{trackNo})$$

The full environment specification is defined by

$$
\begin{aligned}
E = ((\ &|||_{\mathit{trackId},\mathit{trainId}} \mathit{TRAIN}(\mathit{trackId}, \mathit{trainId})) \\
&\underset{\{\mid \mathit{inE}, \mathit{xE}, \mathit{outE} \mid\}}{\|} (\mathit{INsensor}(1) \,|||\, \mathit{Xsensor}(1) \,|||\, \mathit{OUTsensor}(1) \\
&\qquad\qquad |||\, \mathit{INsensor}(2) \,|||\, \mathit{Xsensor}(2) \,|||\, \mathit{OUTsensor}(2))) \\
&\underset{\{\mid \mathit{inE}, \mathit{xE}, \mathit{outE} \mid\}}{\|} (\mathit{Ectrl}1(\langle\,\rangle, \langle\,\rangle, \mathit{OFF}) \,|||\, \mathit{Ectrl}2(\langle\,\rangle, \langle\,\rangle, \mathit{OFF}))
\end{aligned}
$$

Specifying the Target System Test oracles are based on specifications of the target system behaviour. For automatic test evaluation of the TCCC, VVT-RT supports two specification styles: (1) An explicit CSP specification $ASYS$ of the abstract system, written in the same style as the environment specification. (2) Implicit behavioural (timed) trace assertions $(E \underset{I}{\|} SYS) \setminus (\Sigma - I)$ **sat** $S(h)$. In this expression, $(E \underset{I}{\|} SYS) \setminus (\Sigma - I)$ denotes the environment E communicating via interface I with the target system SYS and every event of the complete event space Σ which is not an element of the interface I hidden. $S(h)$ is a predicate with free trace variable h, to be fulfilled by every trace of the target

[8] We use *communication guards*: in an expression $b\&c$, the communication via channel c is refused by the process, if the Boolean expression b evaluates to *false*. $\{\mid c, d, \dots \mid\}$ denotes the set of channel events $\{c.x_1, c.x_2, \dots, d.y_1, d.y_2, \dots\}$, x_i and y_j are values of the channel alphabets of c and d, respectively.

system visible at the system interface I, when running in an environment as specified by E. Specification style (1) is mandatory if VVT-RT is used for the symbolic execution of the target system specification, if the test monitor should determine a measure for the test coverage achieved or if the system is to be applied for automatic *on-the fly* evaluation as *built-in test equipment.* If on-the fly evaluation is unnecessary, several explicit specifications describing different behavioural aspects and referring to different interface channels may be used. The implicit specification style (2) may only be used for offline evaluation of test results.

For the Control Module normal behaviour test configuration, E is the environment specification given above and the interface is defined by

$$I = \{\!|\ in1', x1', \ldots, red', yellow'\ |\!\}$$

For example, an explicit specification only capturing the untimed safety requirements *"whenever a signal is switched ON, the red light must be also ON"* looks like

$$SAFE = SF(false, false, false)$$

$$\begin{aligned} & SF(redIsOn, sig1IsOn, sig2IsOn) = \\ & \quad \neg\ (sig1IsOn \lor sig2IsOn) \& red'?x \rightarrow SF(x = ON, sig1IsOn, sig2IsOn) \\ & \qquad \square\ redIsOn \& sig1'?x \rightarrow SF(redIsSafe, x = ON, sig2IsOn) \\ & \qquad \square\ redIsOn \& sig1'?x \rightarrow SF(redIsSafe, sig1IsOn, x = ON) \end{aligned}$$

An implicit specification stating that the yellow phase should last for at least t_{yellow} time units could be written as

$$\begin{aligned} & S(h) \equiv_{df} \\ & \quad (\exists\, t_1, t_2, s \bullet h \upharpoonright \{\!|\ yellow'\ |\!\} = s^\frown \langle (t_1, yellow'.ON), (t_2, yellow'.OFF) \rangle) \\ & \qquad \Rightarrow t_{yellow} \leq t_2 - t_1 \end{aligned}$$

Validating the Specifications Since the test automation process is completely driven by the environment and target system specifications, their correctness is of great importance. It is not assumed that a formal higher-level document with all the required correctness properties might exist, so that the specification could be formally verified to be a refinement of the higher-level document. As a consequence the specification can only be *validated* by means of various heuristics. To this end, VVT-RT provides three techniques: (1) To validate the untimed safety- and liveness characteristics of the specification, the user can define refinement relations between the explicit specifications E, $ASYS$ and various user-defined auxiliary processes expressing the properties which should be present from the user's point of view. These refinement relations can be automatically verified using the FDR model checker. (2) $(E \underset{I}{\|} ASYS)$ may be symbolically executed using the interactive simulation component of VVT-RT. (3) Explicit specifications of E and $ASYS$ may be validated by creating a collection of redundant implicit specifications $S_i(h)$, that should all be satisfied by $(E \underset{I}{\|} ASYS) \setminus (\Sigma - I)$. Next, the explicit specifications E and $ASYS$ are used to generate traces h_j of $(E \underset{I}{\|} ASYS) \setminus (\Sigma - I)$. Using the second VVT-RT test oracle, these traces are evaluated against the implicit specifications: If $S_i(h_j)$ does not hold for a specific pair S_i, h_j, the explicit specifications E and $ASYS$ violate the assertion $(E \underset{I}{\|} ASYS) \setminus (\Sigma - I)$ **sat** $S_i(h)$, so either $E, ASYS$ or S_i contains a specification error.

4.5 Generating and Driving Test Cases

As indicated above, the VVT-RT system generates and drives test cases using the transition graphs associated with sequential environment processes. Each graph is executed by a separate process. The process determines its local readiness to engage into a certain set of events. A scheduler controls the synchronization requirements and marks all events which are ready for execution to be dispatched. For untimed and timed specifications different schedulers have to be used: The untimed CSP semantics allows events e to be interleaved by an arbitrary number of other events though e could be generated because every communication partner involved is waiting to engage into e . In contrast to this, the real time semantics requires that every event occurs as soon as all communication partners are ready for it and hidden events occur as soon as they are available [4]. This avoids unnecessary distinctions between different sequences of simultaneous events: These sequences are considered as equivalence classes, and one member of this class is sufficient to be covered by the test executions.

The test driver performs *on-the-fly* test generation, executing the transition graphs using a breadth-first strategy, as required for trustworthy failure detection [17]. Since for the test of the TCCC timed trace refinement is an appropriate correctness relation, the test driver operates with less complicated test cases than the untimed failures-divergence test driver specified in [17]: Both target system and environment never refuse inputs, therefore it is unnecessary to analyze the non-blocking properties of the target system for each trace length. Furthermore the test driver can choose any input to the target system but cannot influence the sequence of target system outputs by blocking the corresponding channels.

4.6 Evaluating the Test Results

Given the specification types described above, the implementation of test oracles in the VVT-RT system is straight forward: During test execution, the timed trace of all interface events is recorded. If explicit specifications have been provided, the trace is evaluated against a global (timed) transition graph derived from the specification. In case of failures, the correct prefix of the trace is displayed together with the erroneous event and possible correct alternatives. It is interesting to note that the CSP semantics allows easier evaluation algorithms as for example the CCS semantics [13]: Since CCS distinguishes between late and early branching, several transitions emanating from a state of a CCS transition graph may be labelled by the same event. This is not the case for CSP transition graphs. As a consequence, a test evaluation algorithm for CCS transition graphs has to apply backtracking techniques, since identical traces might be represented by different walks through the graph.

For the evaluation of the timed trace against implicit behavioural assertions, a component of the test tool developed by Hörcher and Mikk [11, 12] for the automatic evaluation of Z specifications is integrated into VVT-RT: Trace assertions are specified as Z schemas containing predicates over a free trace variable h : seq. The Z test tool automatically transforms the predicate parts of Z schemas into executable evaluation functions that are used to check the timed trace observed during a VVT-RT test execution.

5 Conclusion

In this article we have discussed several important aspects of test automation for reactive real-time systems. These aspects were illustrated by examples from tests performed for a tramway crossing control computer which is now in operation since 1995.

The test results were considered as successful [18] for the following reasons:

- Several errors were found by means of the test automation system VVT-RT, after the target system had been thoroughly tested in a manual way and passed the official acceptance tests performed by the certification authorities.
- The number of tests automatically performed and documented by the test system resulted in about 3600 pages of test documentation[9] which would otherwise have to be manually produced to document the same degree of test coverage without tool support.
- Though the test procedure using VVT-RT was performed for the first time and the formal specifications used for testing had to be explicitly produced for this purpose, the total costs of the automized tests were less than 30% of the costs estimated for an equivalent manual test.

Acknowledgements I would like to thank my friends and collaborators for their substantial contributions to the concepts and results presented in this article: Alexander Baer, Bettina Buth, Sabine Dick, Detlef Falkmeyer, Ute Hamer, Erich Mikk, Wolfgang Noack, Anders Ravn, Hans Rischel and Michael Siegel. This work was partially supported by the German Ministry of Education and Research BMBF, project UniForM.

References

1. K. R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs.* Springer-Verlag, Berlin Heidelberg New York (1991).

[9] This corresponds to 98% branch coverage of the transition graph representing the specification.

2. E. Brinksma: A theory for the derivation of tests. In P. H. J. van Eijk — C. A. Vissers and M. Diaz (Eds.): *The Formal Description Technique LOTOS.* Elsevire Science Publishers B. V. (North-Holland), (1989), 235-247.
3. RTCA DO178B: *Development considerations in airborne computer systems.* (1993).
4. J. Davies: *Specification and Proof in Real-Time CSP.* Cambridge University Press (1993).
5. ELPRO LET GmbH: *Programmablaufplan – Bahnübergang.* ELPRO LET GmbH (1994).
6. Formal Systems Ltd.: *Failures Divergence Refinement.* User Manual and Tutorial Version 1.4. Formal Systems (Europe) Ltd (1994).
7. M.-C. Gaudel: Testing can be formal, too. In P. D. Mosses, M. Nielsen and M. I. Schwartzbach (Eds.): *Proceedings of TAPSOFT '95: Theory and Practice of Software Development.* Aarhus, Denmark, May 1995, Springer (1995).
8. M. C. Hennessy: *Algebraic Theory of Processes.* MIT Press (1988).
9. C.A.R. Hoare. *Communicating sequential processes.* Prentice-Hall International, Englewood Cliffs NJ (1985).
10. H. M. Hörcher and J. Peleska: The Role of Formal Specifications in Software Test. Tutorial, held at the FME '94.
11. H. M. Hörcher: Improving Software Tests using Z Specifications. To appear in J. P. Bowen and M. G. Hinchey (Eds.): *ZUM '95: 9th International Conference of Z Users*, LNCS, Springer (1995).
12. E. Mikk: Compilation of Z Specifications into C for Automatic Test Result Evaluation. To appear in J. P. Bowen and M. G. Hinchey (Eds.): *ZUM '95: 9th International Conference of Z Users*, LNCS, Springer (1995).
13. R. Milner: *Communication and Concurrency.* Prentice-Hall International, Englewood Cliffs NJ (1989).
14. M. Müllerburg: Systematic Testing: a Means for Validating Reactive Systems. In *EuroSTAR'94: Proceedings of the 2nd European Intern. Conf. on Software Testing, Analysis&Review.* British Computer Society, (1994).
15. G. J. Myers: *The Art of Software-Testing.* John Wiley & Sons, New York (1979).
16. J. Peleska: *Trustworthy Tests for Reactive Systems — Automation of Real-Time Testing.* In preparation, JP Software-Consulting (1996).
17. J. Peleska and M. Siegel: From Testing Theory to Test Driver Implementation. To appear in *Proceedings of the Formal Methods Europe Conference FME '96*, LNCS, Springer-Verlag, (1996).
18. J. Peleska: Testautomatisierung für diskrete Steuerungen, Anwendung: Bahnübergangssteuerung – Abschlußbericht Phase 1. Technical Report 06/95, JP Software-Consulting (1995).
19. J. Peleska: Formal Methods and the Development of Dependable Systems. Technical Report 07/95, JP Software-Consulting (1995).
20. D. J. Richardson, T. O. O'Malley and C. Tittle Moore: Approaches to Specification-Based Testing. In *ACM SIGSoft 89: Third Symposium on Software Testing, Analysis and Verification*, December (1989).
21. D. J. Richardson, S. Leif Aha and T. O. O'Malley: Specification-based Test Oracles for Reactive Systems. In *Proceedings of the 14th International Conference on Software Engineering, Melborne, Australia*, May (1992).
22. E. Weyuker, T. Goradia and A. Singh: Automatically Generating Test Data from a Boolean Specification. *IEEE Transactions on Software Engineering* Vol. 20, NO. 5, (1994).