Available online at www.sciencedirect.com
ScienceDirect

Electronic Notes in Theoretical Computer Science 260 (2010) 225–241

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Integrating Component Tests to System Tests

Bernhard Schätz Christian Pfaller

Technische Universität München, Fakultät für Informatik, Boltzmannstr. 3, D-85748 Garching bei München, Germany

Abstract

During the verification phase in component-oriented approaches to (embedded) system development component tests are generally followed by system tests, often using different testing specifications and environments. Thus, when integrating components in the overall system, the actual implementation platform may differ from the development platform, leading to a deviation in the component behavior. This raises the issue of testing the functionality of a component integrated in a larger system without the use of an – instrumented – development platform, allowing to perform glass-box-testing.
We show how a test case of a component can be transformed into a black-box test case of the overall system, allowing to deduce the validity of the test case of the component from the validity of the test case of the system. To mechanize the transformation, we provide a formalization of test cases and their integration in component networks in WS1S, and apply the Mona model checker based on this formalization to automatically generate system test cases from component test cases.

Keywords: Test case, generation, component, system, integration, WS1S, model checking

1 Introduction

In many application domains, e.g., automotive systems or production and automation, embedded software has become the driving force on innovation, leading to an increasing demand for software-controled functionality. To meet this increasing demand for software, a component-based approach to the construction of (software) system has become *the* foundation of state-of-the-art software engineering. Besides reducing the complexity of the overall engineering process, component-based software engineering also supports the construction of customized systems by combining pre-fabricated components with others custom-made for the specific system.

In a component-based approach, quality assurance in the constructive phase is usually performed in the form of *component tests*, and *integration* or *system tests*. However, especially in the construction of customized embedded systems like automobiles, quality assurance is faced with two problems, which often occur during system evolution where most components of the system are reused and only a few components are changed or replaced:

doi:10.1016/j.entcs.2009.12.040

- testing of the final implementation should be restricted as much as possible to the changed/custom-made components to efficiently cope with the large number customized systems
- system tests have to be performed on the implementation-level platform, which does not supply glass-box-testing with access to all internal data needed for testing a component without costly or invasive instrumentation of the implementation

Therefore, in the following, an approach is introduced that allows to transform component tests to system tests, supporting the verification of test cases for individual components using only the interface of the overall system. A transformation is not always possible, for a given component test (on component level) an equivalent test case on system level may not exist.

After discussion the context and contribution of the presented approach in the remainder of this section, a formalization of observations about components and networks of components is introduced in Section 2 to provide the basics for the applied techniques. As the main part, Section 3 introduces a method for generating such test cases on a system level from test cases for components. Finally, in Section 4 the presented approach is summarized, and compared to other methods of test case generation.

1.1 Context

In software testing test cases are derived for a certain *system under test (SUT)* with the aim of detecting faults in this SUT. The test cases are defined over the interface of the considered SUT, and the interface of the test execution and observation harness usually reflects this interface of the SUT; thus test cases can by applied directly. This is the standard situation of test case derivation and execution. In some cases the situation is different: Assume that a specific component which is already an integrated part of a larger system (consisting of further components) should be tested. We assume further that there is no direct access to the component's interface from the system interface. Now the question arises how to test the (sub-)component if we only have the possibility to execute test cases and observe its results at the interface of the complete system. To distinguish between the complete system and the considered specific component we use the term *component under test (CUT)* for such a component in the further.

Hence it is the aim of the presented method to do component testing but execute the test cases at the interface of the integrated system—we do not aim on system or integration tests. In practice there are many situations where such settings occur, for example: (1) There is already a test harness for executing tests at the system level, but no test harness or execution environment for test cases on component level is available. (2) A new third-party component (which may be faulty) was integrated in an otherwise unchanged system and there is no direct access to the component interface available. (3) Setting up a specific test execution environment for every single component is to costly and too time consuming, or not possible. Especially, if embedded devices are to be tested on the implementation level control unit, internal

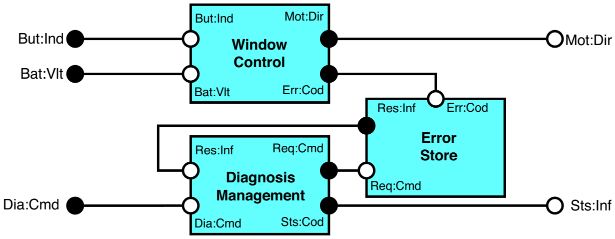

Fig. 1. Power-Window Control Unit: System including Error Store

data within that unit may not be accessible from the outside. Such an approach may be also useful if we (later) know about a certain defect in an integrated sub-component and want to see the effect of that defect to the complete system at the interface of the implementation-level platform. Reuse may also be a motivation for executing component tests on system interface level: Often component tests are already available and these should be translated in a way that they can be executed at the implementation-level – it will be very useful to know which of these test cases actually test the CUT independently from the surrounding components. The other way around there might already exist system tests and we want to know which of these system tests shed specific light on the behavior of a single sub-component.

1.2 Contributions

The introduced method aims at deducing a system test from a component test, allowing to verify that component test case at the system level without instrumentation of the system. To that end, we

- introduce a common formalization of behavioral specifications in the form of the exemplary behavior – like test case descriptions – as well as complete behavior – like component descriptions – including the combination of components to networks.
- define the notion of a test case to be satisfied for a component or system, allowing to check for the validity of a test case.
- define the concept of a system test case verifying a test case for a component of that system.
- provide automatic support to generate a test case for the overall system from a test case of a component, verifying that test case of the component if such a test case exists.

To illustrate the approach, the example of an automotive electronic control unit of a power-controlled window is used. As shown in Figure 1, the control unit holds three software components:

Window Control, which translates a button signal received via port But indicating the direction of the intended movement of the window (Ind = Up, Hd, Dn) into a motor signal sent via Mot indicating the executed movement of the window (Dir = Lo, Zr, Hi), provided the battery signal received via Bat indicating the current

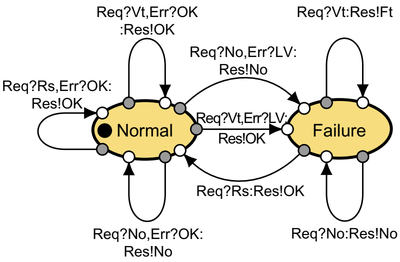

Fig. 2. Behavior of the Error Store Component

voltage (Vlt = Lo, Hi) states sufficient power; an error signal sent via port Err indicating the error code (Cod = LV, OK) provides a low voltage error otherwise.

Error Store, which stores an error signal received via port Err indicating the error code; if requested to return or reset the error status by a request signal received via port Req indicating the command (Cmd = Vt, Rs, No), the result of the request is returned as result signal via port Res indicating the corresponding information (Inf = Ft, OK, No).

Diagnosis Management, which translates a diagnostic signal received via port Dia – indicating the diagnostic command – into a corresponding request signal sent via port Req – indicating the requested command – and forwards the result signal received via port Res – indicating the returned information – as a status signal sent via port Sts – indicating the status information.

Since all three components are packaged and deployed onto the control unit, only the external signals received and sent via Bat, But, Dia, Mot, and Sts are available for testing any software component deployed on the unit. Thus, to check the validity of a test case as shown in Figure 4, verifying the reset-set-query functionality of the Error Store component, a corresponding test case is needed that is immediately executable at the interface of the control unit and is indirectly assuring the validity of the test case of the component.

2 Describing Behavior

In this section, *components* are introduced as building blocks for the construction of reactive systems. The behavior of a component is described as a transition system, controlling the interaction of the component with its environment or surrounding system via input and output signals, and formalized as the set of possible *observations* about the interaction with its environment, described by the history of exchanged signals. Finally, the notion of observations is extended to *systems* described as networks of communicating components.

2.1 Transition Systems

To describe the behavior of a reactive component, often transition systems are used in the form of state-transition-diagrams. Here, the notation and formalism introduced in [12] and [11], resp., is used which allows the modular formalization of clocked, hierarchical transition systems. For reasons of brevity, in the following only non-hierarchical state-transition diagrams are considered.

Figure 2 shows the graphical notation used to describe the transition system of a component for the example of the Error Store component, with

- the set Loc of control locations, used to describe its control state; e.g., the set $Loc = \{\mathsf{Normal}, \mathsf{Failure}\}$
- the set Var of variables, used to describe its data state; e.g., the set $Var = \{\mathsf{Err}, \mathsf{Req}, \mathsf{Res}\}$
- the set $Trans$ of transitions of the form $(a, pre, post, b)$, each consisting of a start location a and end location b from the set of locations Loc, as well as a pre-state pre and post-state $post$ expressions assigning values to variables from Var; e.g., the transition covering the simultaneous occurrence of a voltage error and the empty command is described by the labeled transition from Normal to Failure with a label consisting of Req?No, Err?LV and Res!No

Using these elements, the behavior of a transition system in terms of simple single-step executions is defined. Each transition corresponds to a single step of computation/interaction. When entered through its entry location, it reads the values of its variables; it then changes the variable state by writing new values and terminates by exiting via its exit location.

To describe a transition, we use the notation described in [12]. Using the above example, the first part of the label states that whenever the no-command signal No is received via variable Req and – at the same time – the voltage-error signal LV is received via variable Err, then the transition is enabled. The second part of the label states that, whenever the transition is triggered, in the next state the no-error signal No is sent via variable Res. These parts use a short-hand notation for reading pre-transtion values and writing post-transition values. They correspond to terms $\grave{\mathsf{Err}} = \mathsf{LV}$ and $\mathsf{Res}' = \mathsf{No}$, respectively, using variables $\grave{v}$ with $v \in Var$ for values of v prior to execution of the transition, and variables v' with $v \in Var$ for values of v after its execution.

The formalization of a transition system is based on the assignment of values to variables – for the modeling of the data state – via $\overrightarrow{Var} = Var \to Val$.[1] Abstracting from a concrete graphical representation, a transition is described as the structure (a, t, b) with entry location a, exit location b, and transition label t over $\overrightarrow{Var} \times \overrightarrow{Var}$. t corresponds to the conjunction of the pre- and the post-part of the label. Its behavior is the set of *simple executions* containing all elements

- $(a, \mathit{before} \circ \mathit{after}, b)$
- $(a, \mathit{before} \circ \mathit{after})$
- (a, before)
- $(a, \langle\rangle)$[2]

[1] We assume a universal set Val containing all possible values of variables.

[2] $\langle\rangle$ describes the empty sequence.

with $t = (\mathit{before}, \mathit{after})$. Consequently, the behavior of the above transition is the set consisting of all simple executions $(\mathsf{Normal}, \mathit{before} \circ \mathit{after}, \mathsf{Failure})$, $(\mathsf{Normal}, \mathit{before} \circ \mathit{after})$, $(\mathsf{Normal}, \mathit{before})$, and $(\mathsf{Normal}, \langle\rangle)$, such that $\mathit{before}(\mathsf{Err}) = \mathsf{LV}$ and $\mathit{before}(\mathsf{Req}) = \mathsf{No}$, as well as $\mathit{after}(\mathsf{Res}) = \mathsf{No}$.

Note that transitions need not explicitly assign a value to each of its variables from Var; non-assigned variables in the pre- as well as post-condition are implicitly treated as getting a value nondeterministically assigned. E.g., The request-transition in the Failure-location of Figure 2 with label Req?Vt : Res!Ft corresponds to a set of executions of the above form with $\mathit{before}(\mathsf{Err}) \in \{\mathsf{OK}, \mathsf{LV}\}$.

2.2 Component Behavior

The description of a component capsules the transition system describing its behavior. Graphically, it consists of the description of the transition system, e.g., provided by Figure 2, and by the description of its interface, e.g., provided by the Error Store element in Figure 1.

Therefore, including its transition system, a component is described by

- declaring a (non-empty) subset $\mathit{Init} \subseteq \mathit{Loc}$ of initial control locations; e.g., the location Normal, marked by a black dot in Figure 2
- declaring a (non-empty) subset of $\mathit{InOut} \subseteq Var$ of input and output variables for exchanging signals with the environment; e.g., the input variables Err and Req as well as output variable Res, depicted by empty and filled circles in Figure 1

Based on these additional descriptions, the behavior of a component can be defined. For a state $s : Var \rightarrow Val$ with $Var' \subseteq Var$, the notation $s \uparrow Var'$ is used for restrictions $(s \uparrow Var')(v) = s(v)$ for all $v \in Var'$. This restriction is extended to sequences of states through point-wise application. For sequences r and t we use the notation $r \circ t$ to describe the concatenation of r and t.

First, the set of *executions* of a component is introduced. An execution is either a triple (a, t, b) consisting of a finite sequence $t \in \overrightarrow{\mathit{Var}}^*$ of states corresponding to an execution starting at location a and ending at location b, changing variables according to t; or it is a pair (a, t) consisting of a finite sequence t of states, starting at location a. In the context of testable behavior a restriction to finite observations is sufficient. The set of executions is inductively characterized by all (a, t, b) and (a, t) with

- (a, t, b) and (a, t) are simple executions of the transition system of the component
- $t = t_1 \circ s \circ t_2$ is the concatenation of t_1, s, and t_2 such that $(a, t_1 \circ s, c)$ and $(c, s \circ t_2, b)$ as well as $(c, s \circ t_2)$ are executions of the component for some $b \in \mathit{Loc}$.

Using the example of the Error Store, $(\mathsf{Normal}, s_1 \circ s_2 \circ s_3, \mathsf{Failure})$ with

- $s_1(\mathsf{Err}) = \mathsf{OK}$, $s_1(\mathsf{Req}) = \mathsf{No}$, $s_1(\mathsf{Res}) = \mathsf{No}$
- $s_2(\mathsf{Err}) = \mathsf{OK}$, $s_2(\mathsf{Req}) = \mathsf{Rs}$, $s_2(\mathsf{Res}) = \mathsf{No}$
- $s_3(\mathsf{Err}) = \mathsf{LV}$, $s_3(\mathsf{Req}) = \mathsf{No}$, $s_3(\mathsf{Res}) = \mathsf{OK}$

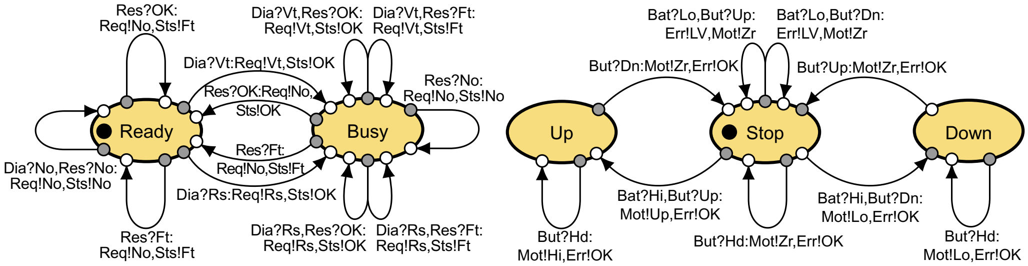

Fig. 3. Behavior of the Diagnosis Management and Window Control Components

is a possible execution of the component.

In a second step, executions are reduced to *observations*. Since observations about a component are restricted to the interface of a component, locations as well as non-input/output-variables have to be removed from the executions to form observations. Therefore, the set of all observations o of a component is defined by the set of all executions (a, t, b) and (a, t) with

- a is an initial interface location of the component, i.e., $a \in \mathit{Init}$[3]
- s is the restriction of t to the component interface, i.e., $s = t \uparrow \mathit{InOut}$
- b is an interface location of the component

Finally, the *behavior* of a component is the set $\mathit{Obs} \subseteq \overrightarrow{\mathit{InOut}}^*$ of all its observations. Again using the example of the Error Store component, $s_1 \circ s_2 \circ s_3$ as defined above is a possible observation of the component, since here $\mathit{InOut} = \mathit{Var}$ and $\mathit{Normal} \in \mathit{Init}$.

2.3 Component Networks

To deal with the issues of system integration described in Section 1, the definition of observations of components must be extended to hierarchical systems, i.e., components which themselves contain subcomponents communicating via common variables. Graphically, networks of components communicating over variables are described by component diagrams as shown in Figure 1. The communication via common variables is indicated by lines, linking output variables to input variables. For each non-hierarchical component its behavior is described by a transition system; e.g., the behaviors of Diagnosis Management and Window Control are shown in Figure 3.

As introduced in Subsection 2.2 in the case of the description of components, the interface InOut_S of a network S in term of its input and output variables is a subset of the combined interfaces InOut_{C_i} of its components C_i. As in the previous case, non-interface variables from $(\bigcup_{i=1,\dots,n} \mathit{InOut}_{C_i}) \backslash \mathit{InOut}_S$ are considered to be hidden variables, used only for internal communication.

Like the interface of a network of components, the *behavior of a network* can be deduced from the behaviors of its components. An observation of the complete system S is obtained from the projection to the behavior of its subcomponents

[3] As an extension, also initial values of output variables may be defined.

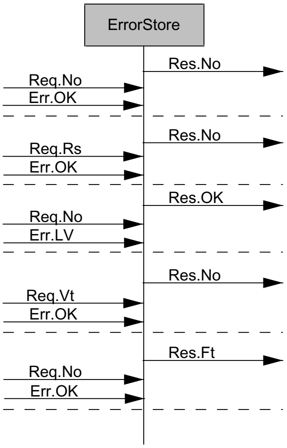

Fig. 4. Reset-Set-Query-Test Case of the Error Store

$C_1, \ldots C_n$. Formally,

$$t \uparrow InOut_S \in Obs_S \Leftrightarrow \bigwedge_{i=1,\ldots,n} t \uparrow InOut_{C_i} \in Obs_{C_i} \qquad (1)$$

3 Generating Tests

In this section, *test cases* are introduced as (sets) of observations, defining the validity of a test case of a component as its containment in behavior of the component. Furthermore, the notion of an *integrated test case* is introduced, demonstrating its ability to test a component behavior of the system level. Finally, its *automatic generation* using model checking is illustrated.

In this section, instead of using the set-based notation $Obs_C \subseteq \overrightarrow{InOut_C}^*$ from Section 2 for the observations of a component C with interface $InOut_C$, an equivalent predicate-based notation $C : \overrightarrow{InOut_C}^* \to \mathbb{B}$ is used via the characteristic predicate

$$C(t) \Leftrightarrow t \in Obs_C$$

3.1 Test Case Specification

Intuitively, a test case describes an observation of a component or system, i.e., a sequence of values sent and received on the interface of a component or systems. Since in Section 2, the behavior of a component or system is described as a set of observations about it, a test case simply is an element of that set. While basically such an element can be described by a simple (linear) transition system, often special diagrams are used.

Graphically, a test case can, e.g., be specified using variations of Message Sequence Charts or Sequence Diagrams. Figure 4 shows the representation of such a test case, using timed *Extended Event Traces* as introduced in [12]. The test case description consists of

- a *labeled life line* described by a vertical line for the component participating in the test case, indicating the sequence of observations from top to bottom; e.g., the life line of the Error Store component
- a set of interactions depicted as labeled arrows describing the exchange of data between the participating component as well as its environment; e.g., receiving the value Rs via variable Req from the environment (or test bed), or sending the value No via variable Res to the environment (or test bed)
- a set of *time regions* delimited by dashed lines collecting simultaneous interactions within those regions; e.g., the region containing the simultaneous receiving of No and LV via Req and Err, resp., as well as the sending of OK via Res

Note that this technique can also be applied to describe the interaction between several components as well as their environment. Figure 5 shows such a description for a sequence of interactions of Error Store, Diagnosis Management and Window Control.

Formally, such a diagram can be described as an observation (or set of observations) about the interface of the participating components; the diagram describes a sequence (or set of sequences) of states, with each state assigning a value to an interface variable according to its corresponding time region. Thus, e.g., Figure 4 describes an observation of length 5, with the third state assigning the values OK, No, and LV to the variables Res, Req, and Err, resp.

In the setting of components with input and output variables as described above, the description of a test case consists of two parts:

(i) the specification of the *input signals* received by the system under test from the environment during the execution of the test

(ii) the specification of the *output signals* sent from the system under test to the environment during the execution of the test

The system under test the satisfies this test case, when the output actually provided by the system corresponds to the output prescribed by the test case, given the system received the input prescribed by the test case. Similar to the description of the components, a predicate-based notation is also used for test case via

$$T(t) \overset{\text{def}}{=} t \uparrow In_C \in Obs_T \uparrow In_C \Rightarrow t \uparrow Out_C \in Obs_T \uparrow Out_C$$

for a set Obs_T characterizing the behavior specified by a test case for a component C with input variables In and output variables Out.[4]

Intuitively, a test case and its graphical representation is interpreted as the set of observations about the component under test that correspond to the prescribed behavior, i.e., with either an input differing from the prescribed input or an output corresponding to the prescribed output. Thus, a test case characterizes all legal behaviors of that component under test; it therefore is a super-set of all the expected observations of the component.

Consequently, a test case is valid for a given component if the set of observations

[4] The restriction $t \uparrow Var$ is extended to restrictions on sets of sequences via element-wise application.

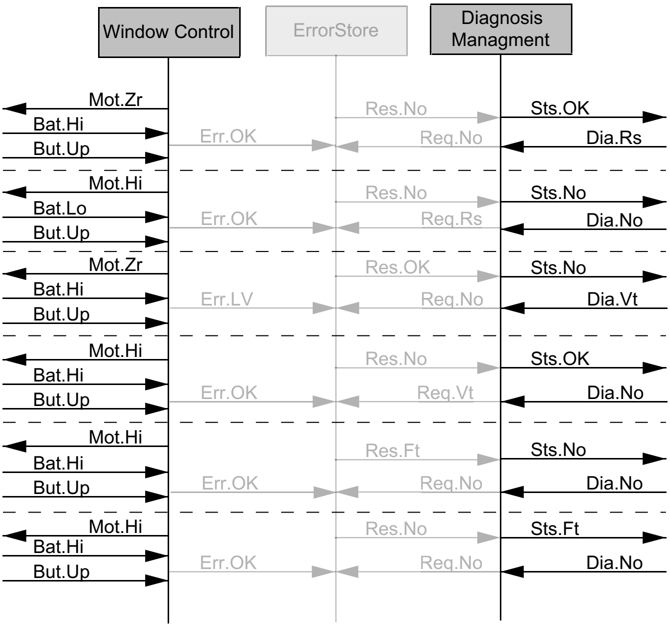

Fig. 5. Composed Observation of ErrorStore, DiagnosisManagement, and WindowControl

corresponding to the test case contains the set of observations corresponding to the behavior of the component under test. If interpreting the description of the test case as well as the behavior of the component as predicates over observations over the alphabet of the component, the following definition of the validity of a test case is obtained.

Definition 3.1 Satisfied Test Case A system or component C *satisfies a test case* T if

$$\forall t \in \overrightarrow{Var_C}^{*}.C(t) \Rightarrow T(t) \tag{2}$$

For example, the sequence diagram in Figure 4 shows a possible behavior of the Error Store specified by the transition system of Figure 2.

3.2 Integrated Test Cases

In the following, components C and B are subcomponents of system A, with interfaces Var_C, Var_B, and Var_A, resp, where $Var_A \subseteq Var_B \cup Var_C$. Furthermore, C is assumed to be completely integrated into A, i.e., having no direct connection to the environment of A; Figure 1 shows an example of such a situation with C corresponding to ErrorStore, B corresponding to DiagnoseManagement and WindowControl, and A corresponding to their composition. The following definition, theorem, and proof can be easily extended to the more general case with C being only partially integrated.

As mentioned in Section 1, during integration the interface of a component C embedded into the overall system might not be directly observable. Therefore, to ensure a certain property given in form of a test case T, indirect observation may be needed. Intuitively, we are looking for test case S for the overall system – consisting of C and the rest of the system B – such that if the system under test fulfills that test case S, also C must fulfill test case T.

This concept of a test case for a component, integrated into a overall system to deduce a test case for the complete system, is captured in the following definition, again interpreting a test case as well as system specification as a predicate over the observations corresponding to the test case and system specification, resp.

Definition 3.2 Integrated Test Case A test case S is a *test case for* T *integrated into* B if[5]

$$S(s) \overset{\text{def}}{=} \forall t \in \overrightarrow{Var_C}^{*}.(B(s,t) \Rightarrow T(t)) \tag{3}$$

For example in the case of the power window implemented by the network of ErrorStore, DiagnoseManagement, and WindowControl as shown in Figure 1, the reset-set-query test case shown in Figure 4 cannot be directly validated since the interface of ErrorStore is completely hidden within the control unit. Any observation on its ports Err, Req, and Res can only be made indirectly over the interface of the control unit.

The explicit characterization of an integrated test case S given by 3, which is especially suited for the application of text case generation as shown in Subsection 3.3, can also be equivalently characterized more implicitly:

$$S(t \uparrow Var_A) \Leftrightarrow (B(t \uparrow Var_B) \Rightarrow C(t \uparrow Var_C) \tag{4}$$

This characterization will be used in the following to verify the intended properties of an integrated test case.

Thus, to verify that Error Store actually satisfies the behavior prescribed by the component test case, a different test case is needed, which is

- ensuring the validity of the test case for Error Store
- relying only on the behavior of Diagnose Management and Window Control, but not on the behavior of Error Store
- reading and writing the values of Err, Req, and Res through the interface of the control unit.

Figure 5 shows the description of such an integrated test case. The greyed-out parts of the test case description correspond to execution of the reset-set-query test case of Error Store, which is not directly observable from the interface of the control unit; the solid parts correspond to the reading and writing of ports at the interface of the unit, performed, e.g., by the test-bed for the control unit.

This is reflected in the life lines of the integrated test case. While the interactions attached to the life line of Error Store correspond to the component test case shown in Figure 4, the interactions attached to the life lines of Diagnose Management and Window Control reflect their corresponding behavior as described by the transition systems in Figure 3.

[5] The notation (r, s) for $r \in \overrightarrow{Var_1}^{*}$ and $s \in \overrightarrow{Var_2}^{*}$ with $Var_1 \cap Var_2 = \emptyset$ is used to denote the sequence t of point-wise combined states with $t \uparrow Var_1 = r$ and $t \uparrow Var_2 = s$.

If the values of the input ports Bat, But, and Dia are written as described by the integrated test case, the behaviors of Diagnosis Management and Window Control ensure that the corresponding values of the internal ports Err and Req are assigned to the values prescribed by the component test case. Symmetrically, the behavior of Diagnose Management (and Window Control) ensures that if the values of the output port Sts (and Mot) are read as described by the integrated test case, the values of the internal port Res are assigned to the values prescribed by the component test case.

A system test executable in a test-bed for the control unit can be obtained from Figure 5 simply by removing the greyed-out part corresponding to the component test. Note that the validity of the system test *ensures* the validity of the component test, as stated in the following theorem.

Theorem 3.3 *Satisfied Integrated Test Case An integrated test case S actually is a positive system test for the component test, i.e., component C satisfies test case T if the system consisting of B and C satisfies S, provided such a test is possible within the context of B.*

The proof of Theorem 3.3 can be preformed immediately using first-order logic.

Proof (Theorem 3.3) Assuming that

- the system under test A consists of B and C
- the system under test A satisfies test case S
- S is a test case for T integrated into B

we then can conclude that

- C satisfies test case T (restricted to the context of B)

by means of

$$\begin{aligned}(2,1) &\Rightarrow \forall t.(B(t \uparrow Var_B) \wedge C(t \uparrow Var_C)) \Rightarrow S(t \uparrow Var_A)\\ (4) &\Rightarrow \forall t.(B(t \uparrow Var_B) \wedge C(t \uparrow Var_C)) \Rightarrow (B(t \uparrow Var_B) \Rightarrow T(t \uparrow Var_C))\\ &\Rightarrow \forall t.(B(t \uparrow Var_B) \wedge C(t \uparrow Var_C)) \Rightarrow T(t \uparrow Var_C)\\ &\Rightarrow \forall t.B(t \uparrow Var_B) \Rightarrow (C(t \uparrow Var_C)) \Rightarrow T(t \uparrow Var_C)\end{aligned}$$

and thus (2) also holds for C and T, within the context of B. □

Note that an integrated test case may not exist, e.g., because B cannot ensure that only input values for C are used as required by T based on input values delivered by the environment; or because B cannot ensure that only output values of C are provided as prescribed by T based on output values delivered to the environment. In such a case, only the trivial solution $S(t) = \mathbb{F}$ exists.

Furthermore, the invalidity of the system test *ensures* the invalidity of the component test, as stated in the following theorem.

Theorem 3.4 *Unsatisfied Integrated Test Case An integrated test case S actually is a negative system test for the component test, i.e., component C does not satisfy test case T if the system consisting of B and C does not satisfy S.*

Again, the proof of Theorem 3.4 can be preformed immediately using first-order logic.

Proof (Theorem 3.4) Assuming that

- the system under test A consists of B and C
- C does satisfy test case T
- S is a nontrivial test case for T integrated into B

we then can conclude that

- the system under test does satisfy test case S

by means of

$$\begin{aligned}
(2) &\Rightarrow \forall t.C(t \uparrow Var_C) \Rightarrow T(t \uparrow Var_C)\\
&\Rightarrow \forall t.(B(t \uparrow Var_B) \wedge C(t \uparrow Var_C) \Rightarrow T(t \uparrow Var_C)\\
&\Rightarrow \forall t.(B(t \uparrow Var_B) \wedge C(t \uparrow Var_C) \Rightarrow (B(t \uparrow Var_B) \Rightarrow T(t \uparrow Var_C))\\
(4,1) &\Rightarrow \forall t.A(t \uparrow Var_A) \wedge S(t \uparrow Var_A)
\end{aligned}$$

and thus (2) also holds for A and S. □

3.3 Implementation

To effectively use the definition of integrated test cases, (automatic) support for the generation of test cases is necessary. Since behavior is defined by (possibly infinite) sets of finite traces, test cases are defined as elements of such sets, and integrated test cases are defined via first-order expressions over those sets, a trace-based formalism is best-suited.

Therefore, here WS1S (weak second order monadic structure with one successor function) is used to implement automatic test case generation. This formalism is, e.g., supported by the model checker *Mona* [8]. Using WS1S, behavior is specified by predicates over observation traces, which are – as defined in Subsection 2.2 – sequences of states, i.e. sequences of assignment of values to variables.

Intuitively, these predicates are built up from the most elementary propositions about observations, declaring that a variable has a certain value at a specific point in time in a observation. E.g., looking at the observation corresponding to the reset-set-query-test case in Figure 4, a suitable proposition is "*variable Req has value Vt in the* fourth *state*". Using this approach, a trace can be precisely described by characterizing the states, for which a certain variable has a specific value. E.g., in the above example, a part of the description is characterized by "*variable Req has value No in states 0,2,4, value Rs in state 1, and value Vt in state 3*" (numbering states beginning with 0).

Using this approach, predicates are defined by characterizing a trace via the sets of state indices, for which these basic propositions – linking variables and values – hold. WS1S provides variables (and quantors) over sets of indices, called second-order variables (or quantors) in contrast to zero-order and first-order variables (quantors) for Booleans or indices. Using these second-order variables for sets corresponding to all combinations of variables and values, the above test case can be formalized as

```
pred ErrorStoreTest(
   var2 ErrOK, var2 ErrLV,
   var2 ReqNo, var2 ReqVt, var2 ReqRs,
   var2 ResNo, var2 ResOK, var2 ResFt) =
     (0 in ErrOK & 0 in ReqNo & 0 in ResNo) &
     (1 in ErrOK & 1 in ReqRs & 1 in ResNo) &
     (2 in ErrLV & 2 in ReqNo & 2 in ResOK) &
     (3 in ErrOK & 3 in ReqVt & 3 in ResNo) &
     (4 in ErrOK & 4 in ReqNo & 4 in ResFt);
```

with `var2` declaring second-order parameters, and e.g., `ReqNo` corresponding to the set of indices characterizing states with Req = No.[6] Using the same principle, also the observations of transition-systems can be formalized in a similar fashion, leading to corresponding predicates for `WinCtrl` and `DiagMgt` for the WindowControl and DiagnosisManagement components.

The definition 3.2 of an integrated test case in Section 3.2 can be directly implemented in WS1S, allowing an automatic construction of the corresponding trace using the witness functionality provided by *Mona*:

```
pred SystemTest(
   var2 BatLo, BatHi,
   var2 ButDn, ButHd, ButUp,
   var2 DiaVt, DiaRs, DiaNo,
   var2 MotLo, MotZr, MotHi,
  var2 StsNo, StsOK, StsFt) =
    all2 ErrOK, ErrLV:
    all2 ReqNo, ReqVt, ReqRs:
    all2 ResNo, ResOK, ResFt: ((
    WinCtrl(BatLo, BatHi, ButDn, ButHd, ButUp, MotLo, MotZr, MotHi,
       ErrOK, ErrLV) &
   DiagMgt(RqSRd, RqSBs, DiaNo, DiaVt, DiaRs, ResNo, ResOK, ResFt,
       ReqNo, ReqVt, ReqRs, StsNo, StsOK, StsFt))
    =>  ErrorStoreTest(ErrOK, ErrLV, ReqNo, ReqVt, ReqRs,
       ResNo, ResOK, ResFt));
```

Using the above predicate `SystemTest`, *Mona* can check for its satisfiability, returning the following witness:

```
BatLo = {1} BatHi = {0,2,3,4,5}
ButDn = {} ButHd = {} ButUp = {0,1,2,3,4,5}
DiaVt = {2} DiaRs = {0} DiaNo = {1,3,4,5}
MotLo = {} MotZr = {0,2} MotHi = {1,3,4,5}
```

[6] For reasons of readability disjointness assertions like `ReqNo inter ReqRs` ensuring unique signal values are skipped here.

```
StsNo = {1,2,4} StsOK = {0,3} StsFt = {5}
```

This corresponds to the observation in Figure 5, using the notions for sequence diagrams.

4 Conclusion

The previous sections gave a detailed account of the basics and techniques supporting an automated method for integrating component tests into system tests. In this section, a short summary of the presented approach and an outlook into other possible areas of application of the basic techniques are given. Furthermore, it is set into perspective to other work on automatic test case generation.

4.1 Summary and Outlook

In the previous sections, an approach was introduced for the automatic generation of test cases for integrating a component into a larger overall system. This approach allows to combine a given test case for a component of a system with the behavioral description of the rest of the system, to analyse the behavior of that component without direct access to the interface of the component. This combination delivers a test case executable at the interface of the overall system, which verifies that the component satisfies the given component test case in case that the overall system satisfies the resulting system test case. A typical area for the application of such an approach is, e.g., the construction of embedded software; here, on the implementation level often no means are provided of directly observing the interface of a component integrated in a larger system. This approach is especially useful in the context of system evolution, when only the component under test has been altered, leaving the rest of the system unchanged.

Note that instead of deducing an observation about the overall system from a given observation of a component plus the remainder of the system, also the opposite approach – deducing the required component behavior from the observed system behavior – has its merits in the context of system evolution. This is especially helpful in the context of fault localisation, e.g., when trying to identify possible faulty behavior of a changed component that can result in a faulty behavior at system level, identified during the testing of the otherwise unchanged system. To solve that problem of deducing a component observation from a system observation as well as the behavioral specification of the rest of the system, the same formalizations and techniques can be applied.

4.2 Related Approaches

In this paper we showed a further application for using a model checker in test case derivation – here we generated tests on the system level for testing a sole component. On component level, model-based test case generation by model checking is well-known: In [2] the *SPIN* model checker is used to verify execution traces and to generate further test cases. Ammann et al. generate test cases by using the *SMV* model checker on mutants of the original specification [1]. In [10] model checking is used to generate test cases fulfilling structural coverage on the test model. A similar approach is presented in [4]. For test case generation in general various further techniques exist, as for example using constraint logic programing [9]. Regardless what specific technique is used, to our knowledge all model-based test generation techniques so far assume that the test model (from which test cases are generated) is given at the interface level of the system under test *and* the SUT's interface is also the interface for test case execution. In contrast to these approaches the presented method in this paper uses test cases over the component interface and extends this test cases to the interface of a whole system in which the component is integrated in. Using model checking we are able to decide if a test case is able to unambiguously verify intended behavior or detect errors caused by the component under test (and not by other parts of the system).

The described technique in this paper is focused on supporting issues of integration testing. Other approaches on integration testing include methods based on coverage criteria for integration testing like [5] (data-flow coverage) or [7] (coverage based on component coupling). In [3] mutation testing by mutants of the coupling on the internal interfaces is described as a method for integration testing. [6] gives an overview of various techniques of component testing which are able to be applied by the component user during system integration.

References

[1] Ammann, P., P. Black and W. Majurski, *Using model checking to generate tests from specifications*, Formal Engineering Methods, 1998. Proceedings. Second International Conference on (1998), pp. 46–54.

[2] Callahan, J., F. Schneider and S. Easterbrook, *Automated software testing using modelchecking*, in: *Proceedings 1996 SPIN Workshop*, 1996, also WVU Technical Report #NASA-IVV-96-022.

[3] Delamaro, M., J. Maidonado and A. Mathur, *Interface mutation: an approach for integration testing*, Software Engineering, IEEE Transactions on **27** (2001), pp. 228–247.

[4] Hamon, G., L. de Moura and J. Rushby, *Generating efficient test sets with a model checker*, in: *Software Engineering and Formal Methods, 2004. SEFM 2004. Proceedings of the Second International Conference on*, 2004, pp. 261–270.

[5] Harrold, M. and M. Soffa, *Selecting and using data for integration testing*, Software, IEEE **8** (1991), pp. 58–65.

[6] Jaffar-ur Rehman, M., F. Jabeen, A. Bertolino and A. Polini, *Software component integration testing: A survey*, Technical Report 2005-TR-41, Consiglio Nazionale delle Ricerche, Istituto di Scienza e Technologie dell'Informazione "Alessandro Faedo" (ISTI), Pisa, Italy (2005).

[7] Jin, Z. and A. Offutt, *Integration testing based on software couplings*, Computer Assurance, 1995. COMPASS '95. 'Systems Integrity, Software Safety and Process Security'. Proceedings of the Tenth Annual Conference on (1995), pp. 13–23.

[8] Klarlund, N. and A. Møller, "MONA Version 1.4 User Manual," BRICS, Department of Computer Science, University of Aarhus (2001).

[9] Pretschner, A. and H. Lötzbeyer, *Model Based Testing with Constraint Logic Programming: First Results and Challenges*, in: *Proc. 2nd ICSE Intl. Workshop on Automated Program Analysis, Testing and Verification (WAPATV'01), Toronto*, 2001.

[10] Rayadurgam, S. and M. Heimdahl, *Coverage based test-case generation using model checkers*, in: *Engineering of Computer Based Systems, 2001. ECBS 2001. Proceedings. Eighth Annual IEEE International Conference and Workshop on the*, 2001, pp. 83–91.

[11] Schätz, B., *Modular Functional Descriptions*, in: *Proceedings 4th International Workshop on Formal Aspects fo Component Software (FACS'07)*, Nice, 2007, to appear in ENTCS.

[12] Schätz, B. and F. Huber, *Integrating formal description techniques*, in: J. M. Wing, J. Woodcock and J. Davies, editors, *FM'99* (1999), lNCS 1709.