Test Case Integration: From Components to Systems

Bernhard Schätz and Christian Pfaller

fortiss GmbH
Guerickestr. 25, 80805 München, Germany
{schaeetz,pfaller}@fortiss.org

Abstract: In a component-based development approach system integration generally implies the packaging of software components on hardware units, hiding the interface of a component inside the unit. Thus, integration testing is done on the (logical) component as well as the (technical) unit level, to ensure the correctness of each component in isolation as well as of the complete system combined with the implementation platform. However, analysing the correct functionality of a bundled and deployed component requires direct access to the interface of the component. Here, integrating a component test case into the behavior of the remaining system provides means to verify the functionality of a component from the interface of the unit without additional instrumentation. We show how to mechanically transform a test case of a component into a black-box test case of the overall system.

1 Introduction

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, often using different testing specifications and environments. 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.

Especially in the construction of customized embedded systems, quality assurance is faced with two problems, when most components of the system are reused and only a few components are changed or replaced: 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. Thus 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, as for a given component test an equivalent test case on system level may not exist.

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 be 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: 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; 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; 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 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. To illustrate the approach, the example of an automotive electronic control unit of a power-controlled window is used, as shown in Figure 1:

Window Control translates a button signal But indicating the direction of the intended movement of the window (Ind = Up, Hd, Dn) into a motor signal Mot indicating the executed movement of the window (Dir = Lo, Zr, Hi), provided battery signal Bat indicating the current voltage (Vlt = Lo, Hi) states sufficient power; error signal Err indicating the error code (Cod = LV, OK) provides a low voltage error otherwise.

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

Diagnosis Management translates a diagnostic signal Dia – indicating the diagnostic command – into a corresponding request signal Req – indicating the requested command – and forwards the result signal Res – indicating the returned information – and a status signal 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.
In this section, components are introduced as building blocks for the construction of reactive systems. To describe their behavior, transition systems are used. Here, the notation and formalism introduced in [Sch07], 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.

2 Describing Behavior

In this section, components are introduced as building blocks for the construction of reactive systems. To describe their behavior, transition systems are used. Here, the notation and formalism introduced in [Sch07], 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.

2.1 Transition Systems

Figure 2 shows the notation used to describe the transition system of a component for the example of the Error Store component. Here the set Loc of control locations, describes its control state; e.g., the set Loc = {Normal, Failure}; the set Var of variables, describes its data state; e.g., the set Var = {Err, Req, Res}; Furthermore, the set Trans describes the 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 [Sch07]. 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-transition values and writing post-transition values. They correspond to terms $\text{Err} = \text{LV}$ and $\text{Res} = \text{No}$, respectively, using variables $v$ with $v \in \text{Var}$ for values of $v$ prior to execution of the transition, and variables $v'$ with $v \in \text{Var}$ for values of $v$ after its execution.

The formalization of a transition system is based on the assignment of values to variables modeling of the data state via $\text{Val} = \text{Var} \to \text{Val}$. Abstracting from a concrete syntax, a transition is described as the structure $(a, t, b)$ with entry location $a$, exit location $b$, and transition label $t$ over $\text{Var} \times \text{Var}$. $t = (\text{before}, \text{after})$ corresponds a conjunction of the pre- and the post-part of the label. Its behavior is the set of simple executions containing all elements $(a, \text{before} \circ \text{after}, b), (a, \text{before} \circ \text{after}), (a, \text{before})$, and $(a, \{} \}^2$. Thus, the behavior of the above transition is the set of all simple executions $(\text{Normal}, \text{before} \circ \text{after}, \text{Failure})$, $(\text{Normal}, \text{before} \circ \text{after})$, $(\text{Normal}, \text{before})$, and $(\text{Normal}, \{\})$, such that $\text{before}(\text{Err}) = \text{LV}$ and $\text{before}(\text{Res}) = \text{No}$, as well as $\text{after}(\text{Res}) = \text{No}$.

Note that transitions need not explicitly assign a value to each of its variables from $\text{Var}$; 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 $\text{Req} \circ \text{Val} : \text{Res}! \text{Ft}$ corresponds to a set of executions of the above form with $\text{before}(\text{Err}) \in \{\text{OK}, \text{LV}\}$.

2.2 Component Behavior

The description of a component capsules the transition system. 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 $\text{Init} \subseteq \text{Loc}$ of initial control locations; e.g., the location Normal, marked by a black dot in Figure 2; and furthermore declaring a (non-empty) subset of $\text{InOut} \subseteq \text{Var}$ 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.

For the subsequent formalization, for a state $s : \text{Var} \to \text{Val}$ with $\text{Var}' \subseteq \text{Var}$, the notation $s \circ \text{Val}'$ is used for restrictions $(s \circ \text{Var}')(v) = s(v)$ for all $v \in \text{Var}'$. It is extended to sequences of states through point-wise application. For sequences $r$ and $t$ we furthermore 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 $t \in \text{Var}^*$ 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 $b \in \text{Loc}$.

\footnote{We assume a universal set $\text{Val}$ containing all possible values of variables.}

\footnote{$\{}$ describes the empty sequence.}
Using the example of the Error Store, \((\text{Normal}, s_1 \circ s_2 \circ s_3, \text{Failure})\) with \(s_1(\text{Err}) = \text{OK}, s_1(\text{Req}) = \text{No}, s_1(\text{Res}) = \text{No}, s_2(\text{Err}) = \text{OK}, s_2(\text{Req}) = \text{Rs}, s_2(\text{Res}) = \text{No}, s_3(\text{Err}) = \text{LV}, s_3(\text{Req}) = \text{No}, s_3(\text{Res}) = \text{OK}\) is a possible execution of the component.

In a second step, executions are reduced to observations. Since observations about a component are restricted to its interface, locations and non-input/output-variables are removed from the executions to form observations. Thus, the set of all observations \(s\) 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., \(a \in \text{Init}\); \(s\) is the restriction of \(t\) to the component interface, i.e., \(s = t[@\text{InOut}]\); \(b\) is an interface location of the component. Finally, the behavior of a component is the set \(\text{Obs} \subseteq \text{InOut}^*\) 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 \(\text{InOut} = \text{Var}\) and \(\text{Normal} \in \text{Init}\).

### 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 \(\text{InOut}_S\) of a network \(S\) in term of its input and output variables is a subset of the combined interfaces \(\text{InOut}_{C_i}\) of its components \(C_i\). As in the previous case, non-interface variables

---

Figure 3: Behavior of the Diagnosis Management and Window Control Components

---

3 As an extension, also initial values of output variables may be defined.
Figure 4: Reset-Set-Query-Test Case of the Error Store

from $\bigcup_{i=1,...,n} \text{InOut}_{C_i} \setminus \text{InOut}_S$ 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 $C_1, \ldots, C_n$. Formally, $t@\text{InOut}_S \in \text{Obs}_S \iff \bigwedge_{i=1,...,n} t@\text{InOut}_{C_i} \in \text{Obs}_{C_i}$

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. Instead of using the set-based notation $\text{Obs}_C \subseteq \text{InOut}^{++}_C$ from Section 2 for the observations of a component $C$ with interface $\text{InOut}_C$, an equivalent predicate-based notation $C : \text{InOut}_C^{++} \rightarrow \mathbb{B}$ is used via the characteristic predicate $C(t) \iff t \in \text{Obs}_C$

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 a timed variant of sequence diagrams. 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. 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.

With components with input and output variables as described above, the description of a test case consists of two parts: The specification of the input signals received by the SUT from the environment; the specification of the output signals sent from the SUT to the environment. The SUT 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

$$T(t) \overset{\text{def}}{=} t \otimes \text{In}_C \in \text{Obs}_T \otimes \text{In}_C \Rightarrow t \otimes \text{Out}_C \in \text{Obs}_T \otimes \text{Out}_C$$

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 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

$$\forall t \in \text{Var}_C \cdot C(t) \Rightarrow T(t)$$

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 \(\text{Var}_C, \text{Var}_B\), and \(\text{Var}_A\), resp, where \(\text{Var}_A \subseteq \text{Var}_B \cup \text{Var}_C\). 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,

\(^4\)The restriction \(t \otimes \text{Var}\) is extended to restrictions on sets of sequences via element-wise application.
During integration the interface of a component $C$ embedded into the overall system might not be directly observable. 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 SUT 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 below, 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\footnote{The notation $(r, s)$ for $r \in \overrightarrow{Var}_1$ and $s \in \overrightarrow{Var}_2$ with $\overrightarrow{Var}_1 \cap \overrightarrow{Var}_2 = \emptyset$ is used to denote the sequence $t$ of point-wise combined states with $t \land \overrightarrow{Var}_1 = r$ and $t \land \overrightarrow{Var}_2 = s$.} $S(s) \overset{\text{def}}{=} \forall t \in \overrightarrow{Var}_C^+. (B(s, t) \Rightarrow T(t))$

For example in the case of the power window implemented by the network of ErrorStore, DiagnosisManagement, 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 3.2, which is especially suited for the application of text case generation as shown in Subsection 4, can also be equivalently characterized more implicitly:

$S(t \land \overrightarrow{Var}_A) \iff (B(t \land \overrightarrow{Var}_B) \Rightarrow C(t \land \overrightarrow{Var}_C))$

Figure 5: Composed Observation of ErrorStore, DiagnosisManagement, and WindowControl
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.

If the values of the input ports Bat, But, and Dia are written as described by the integrated test case, the behaviors of Diagnose 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 below:

**Theorem 3.1 (Satisfied Integrated Test Case)** An integrated test case $S$ is a positive system test for a 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 for this and the following theorem can be preformed immediately using first-order logic. Note that an integrated test case may not exist. In such a case, only the trivial solution $S(t) = \emptyset$ exists. Furthermore, the invalidity of the system test ensures the invalidity of the component test, as stated in the following theorem.

**Theorem 3.2 (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 $S$.

### 4 Implementation

To provide tool-support for the generation of integrated test cases, the model of component test cases must be implemented in a way that enables the mechanization of their integration using a suitable framework. In the following, the use of a symbolic model checker for the implementation and mechanization is demonstrated.

#### 4.1 WS1S Formalization

To effectively use the definition of integrated test cases, support for the generation of test cases is needed. As 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. Here WS1S (weak second order monadic structure with one successor function) is used for automatic test case generation. This formalism is supported by the model checker Mona [KM01]. Using WS1S, behavior is specified by predicates over observation traces, which are – as defined above – 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 an 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

\[
\text{pred ErrorStoreTest(} \\
\quad \text{var2 ErrOK, ErrLV, ReqNo, ReqVt, ReqRs, ResNo, ResOK, ResFt) =} \\
\quad (0 \text{ in ErrOK \& 0 \text{ in ReqNo \& 0 \text{ in ResNo) \&}} \\
\quad (1 \text{ in ErrOK \& 1 \text{ in ReqRs \& 1 \text{ in ResNo) \&}} \\
\quad (2 \text{ in ErrLV \& 2 \text{ in ReqNo \& 2 \text{ in ResOK) \&}} \\
\quad (3 \text{ in ErrOK \& 3 \text{ in ReqVt \& 3 \text{ in ResNo) \&}} \\
\quad (4 \text{ in ErrOK \& 4 \text{ in ReqNo \& 4 \text{ in ResFt));}}
\]

with var2 declaring second-order parameters, and e.g., ReqNo corresponding to the set of indices characterizing states with Req = No.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 Diagnose-Management components. The definition 3.2 of an integrated test case in Section 3 can be directly implemented in WS1S, allowing an automatic construction of the corresponding trace using the witness functionality provided by Mona:

\[
\text{pred SystemTest(} \\
\quad \text{var2 BatLo, BatHi, ButDn, ButHd, ButUp, DiaVt, DiaRs, DiaNo,} \\
\quad \text{MotLo, MotZr, MotHi, StsNo, StsOK, StsFt) =} \\
\quad \text{all2 ErrOK, ErrLV ReqNo, ReqVt, ReqRs, ResNo, ResOK, ResFt: } ((} \\
\quad \text{WinCtrl(BatLo, BatHi, ButDn, ButHd, ButUp, DiaVt, DiaRs, DiaNo,} \\
\quad \text{MotLo, MotZr, MotHi, ErrOK, ErrLV) \&} \\
\quad \text{DiagMgt(RqSRd, RqSBs, DiaNo, DiaVt, DiaRs, ResNo, ResOK,} \\
\quad \text{ResFt, ReqNo, ReqVt, ReqRs, StsNo, StsOK, StsFt))} \\
\quad \Rightarrow \text{ErrorStoreTest(ErrOK, ErrLV, ReqNo, ReqVt, ReqRs,} \\
\quad \text{ResNo, ResOK, ResFt));}
\]

4.2 Observation Synthesis

To effectively generate an observation, the witness functionality of model checkers can be exploited. By use of this functionality, the model checker selects an element from a satisfiable, i.e., non-empty model. In case of WS1S, Mona selects a shortest observation.

\[6\text{For reasons of readability disjointness assertions like ReqNo inter ReqRs ensuring unique signal values are skipped here.}\]
satisfying the predicate characterizing the integrated test case. For example, using the witness functionality of *Mona* and applying it to the example of the Error Store component, the sequence diagram in Figure 4 can be synthesized. Using the above predicate SystemTest, *Mona* can check for its satisfiability, returning the following witness:

\[
\begin{align*}
&\text{BatLo} = \{1\}, \text{BatHi} = \{0, 2, 3, 4, 5\} \\
&\text{ButDn} = \{\}, \text{ButHd} = \{\}, \text{ButUp} = \{0, 1, 2, 3, 4, 5\} \\
&\text{DiaVt} = \{2\}, \text{DiaRs} = \{0\}, \text{DiaNo} = \{1, 3, 4, 5\} \\
&\text{MotLo} = \{\}, \text{MotZr} = \{0, 2\}, \text{MotHi} = \{1, 3, 4, 5\} \\
&\text{StsNo} = \{1, 2, 4\}, \text{StsOK} = \{0, 3\}, \text{StsFt} = \{5\}
\end{align*}
\]

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

4.3 Integration Automation

Test case integration consists of two steps: First, the specification of a test case for the CUT – e.g., given by a sequence diagram – is integrated into the specification of the rest of the system - e.g., given by an architectural description as well as the behavior of the remaining components – using the approach described in the previous subsection. Then, the instantiation of the obtained integrated test case is verified w.r.t. to the actual implementation. To illustrate the application, the integration of a changed Error Store component in an otherwise unchanged Power-Window Control Unit system is used. From these specifications, the corresponding WS1S formalizations can be automatically generated as presented above. Combined with the integrated test case, the complete specification is handed over to the *Mona* model checker for processing. If a non-trivial solution exists, a suitable shortest test case is generated; otherwise, the user is informed that no suitable test case can be generated since the specification of the integrated test case is unsatisfiable.

Once the integrated test case has been generated, the behavior of the overall system – containing the component under test – has to be verified with respect to the test case, to show whether the overall system – and therefore also the component under test – respects the prescribed behavior. Figure 2 shows the actually implemented behavior of the Error Store component. It deviates from the intended behavior in the output generated during the change from state Normal to state Failure: while the intended behavior is the provision of an empty result Res!No, the actual behavior is the provision of the OK result Res!OK.

To check whether the implemented system actually satisfies the integrated test case, the implementation – e.g., in form of an electronic control unit – is embedded into test environment allowing to write the inputs and read the outputs at the system interface and thus to execute the – instantiated – test case. To instantiate the test case, its abstract input signals – e.g., the diagnostic command signals Vt, Rs, No – are translated in corresponding concrete binary signals – e.g., 0x00, 0x01, 0x02 – and each round sent to the implementation via the test bed. Similarly, the concrete binary signal received each round are translated back to abstract signals.

Comparing the abstract output values obtained through the test environment with the outputs prescribed by the integrated test case shows whether the system under test satisfies the test case. In case of the defect Error Store component, the observation shown in Figure 6 is obtained, demonstrating that the SUT does not satisfy the integrated test case.

5 Conclusion

The introduced approach enables the automatic generation of test cases for integrating a component into a larger overall system. 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 if 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, e.g., in the context of fault localisation 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, the same formalizations and techniques can be applied.

While the infrastructure of symbolic model checkers often does not scale for the analysis of systems composed from several components, the approach proves to be feasible in the domain of reactive body control using enumeration data types for signal and state variables. Since the tool-supported method allows a stepwise construction of the integrated test case, avoiding an explicit construction of the complete behavior of the remaining system, only the relevant ‘cone of behavior’ is built from the integrated test case.

**References**
