Executing Abstract Test Cases[*]

Bernhard Peischl   Martin Weiglhofer

Franz Wotawa[†]

{peischl, weiglhofer, wotawa}@ist.tugraz.at

Abstract: Generally, test cases derived from a formal model can not be directly fed into implementations under test (IUT), because model based test generation techniques produce abstract test cases. In order to run an abstract test case against an IUT the abstract test case either has to be transformed to a concrete test case or an execution of the abstract test case is needed. In this paper we propose a rule based test execution framework, which allows the execution of abstract test cases. Furthermore, we present first results from testing a so called SIP Registrar by executing abstract test cases derived with the TGV tool from a formal specification.

1   Introduction

Applying model based testing techniques to industrial sized systems requires an appropriate formal model of the system under test. Typically, a formal model abstracts from real world problems in order to get models of manageable size in terms of their state spaces. Due to these abstractions, the output of test generation tools are so called abstract test cases, which describe test scenarios in an abstract way. For the execution of such abstract test cases every stimuli i has to be converted to a concrete message $\gamma(i)$, while a system response o has to be mapped to the abstract level $\alpha(o)$. Normally, manually written test drivers are responsible for the transformation of test messages. However, the implementation of a test driver is a tedious and error prone task.

In difference to our previous work on test generation [APWWa, APWWb], this article deals with execution of the derived test cases. We propose a rule based system for defining γ and α. We show how to execute abstract test cases in terms of our proposed framework and illustrate first experimental results using this novel technique.

This paper continues as follows: in Section 2 we briefly introduce IO-labeled transition systems (IOLTS) and IOLTS test cases. Section 3 presents our test execution approach. In Section 4 we discuss first results. Finally, in Section 5 we present our conclusions.

2 Preliminaries

Since we use TGV [JJ05] for the generation of our test cases the derived test cases are IOLTSs. For details about the underlying testing theory we refer to [Tre96].

Definition 1 (Input Output LTS (IOLTS)) *An IOLTS is a labeled transition system (LTS)* $M = (Q^M, A^M, \to_M, q_0^M)$ *with* Q^M *a finite set of states,* A^M *a finite alphabet (the labels) partitioned into three disjoint sets* $A^M = A_I^M \cup A_O^M \cup \{\tau\}$ *where* A_I^M *and* A_O^M *are input and output alphabets and* $\tau \notin A_I^M \cup A_O^M$ *is an unobservable, internal action,* $\to_M \subseteq Q^M \times A^M \times Q^M$ *is the transition relation and* $q_0^M \in Q^M$ *is the initial state.*

We use the following classical notations of LTSs for IOLTSs. Let $q, q', q_1, \ldots, q_n \in Q^M, Q \subseteq Q^M, a, a_1, \ldots, a_n \in A_I^M \cup A_O^M$ and $\sigma \in (A_I^M \cup A_O^M)^*$. Then, $q \xrightarrow{a}_M q' =_{df}$ $(q, a, q') \in \to_M$ and $q \xrightarrow{a}_M =_{df} \exists q' : (q, a, q') \in \to_M$, and $q \not\xrightarrow{a}_M =_{df} \not\exists q' : (q, a, q') \in \to_M$. $q \xRightarrow{\epsilon} q' =_{df} ((q = q') \vee (q \xrightarrow{\tau}_M q_1 \wedge \cdots \wedge q_{n-1} \xrightarrow{\tau}_M q'))$ and $q \xRightarrow{a} q' =_{df} \exists q_1, q_2 :$ $q \xRightarrow{\epsilon}_M q_1 \xrightarrow{a}_M q_2 \xRightarrow{\epsilon}_M q'$ which generalizes to $q \xRightarrow{a_1 \ldots a_n} q' =_{df} \exists q_0, \ldots, q_n : q = q_0 \xRightarrow{a_1}_M$ $q_1 \ldots q_{n-1} \xRightarrow{a_n}_M q_n = q'$. We denote $q \textbf{ after}_M \sigma =_{df} \{q' | q \xRightarrow{\sigma}_M q'\}$.

Among others a test case $T = (Q^T, A^T, \to_T, q_o^T)$ has following two properties [APWWa, JJ05]: (1) A test case is input complete in all states where an input is possible: $\forall q \in Q^T : (\exists a \in A_I^T, q \xrightarrow{a}_T \Rightarrow \forall b \in A_I^T, q \xrightarrow{b}_T)$. (2) T is controllable: no choice is allowed between two outputs or between inputs and outputs: $\forall q \in Q^T, \forall a \in A_O^T : q \xrightarrow{a}_T \Rightarrow \forall b \in A_I^T \cup A_O^T \setminus \{a\} : q \not\xrightarrow{b}_T$.

3 Abstract Test Execution

Executing abstract test cases consists of two different tasks. One task is test control, i.e., select the next test message and determine the test verdict. The other task is the transformation of the abstract test messages to concrete stimuli and the conversion of concrete system responses to abstract test messages.

The test control procedure for an IOLTS is given by the algorithm of Figure 1. This algorithm takes an IOLTS test case TC and returns a verdict when executing the test case against an implementation under test (IUT). The two methods send(message) (line 6) and receive() (line 9) denote the communication with the IUT. This two methods encapsulate the refinement of requests and the abstraction of responses, respectively. Starting at the initial state q_0^{TC} (line 3) of the test case our test execution algorithm traverses through a certain IOLTS test case until the current state q_c is a verdict state (line 4). If the current state q_c has an output edge (line 5), the stimuli which is represented by the label of the edge is sent to the IUT (line 6). Otherwise, the algorithm waits for a response of the IUT (line 9). If it receives a response, it updates the current state q_c (line 10). Note, that the controllability property (see Section 2) of test cases simplifies the test control algorithm, because the algorithm does not need to handle inputs and outputs at the same state.

```
1   procedure testControl(⟨Q^{TC}, A_I^{TC} ∪ A_O^{TC}, →_{TC}, q_0^{TC}⟩): verdict
2   begin
3     q_c = q_0^{TC};
4     while q_c ∉ {fail, pass, inconclusive} do
5       if ∃p ∈ Q^{TC}, l ∈ A_O^{TC} : (q_c, l, p) ∈→_{TC} then
6         send(l);
7         q_c = p;
8       else
9         msg = receive();
10        q_c = q_c after_{TC} msg;
11      fi
12    done
13    Result := q_c;
14  end
```

Figure 1: Test control algorithm for IOLTS test cases.

For the transformation of messages, we use a rule system $\mathcal{R}_S$. $\mathcal{R}_S$ consists of two sets of rules γ_S and α_S, a set of mappings of abstract to concrete values $\mathcal{M}_S$, and a set of variables $\mathcal{V}_S$:

$$\mathcal{R}_S = \langle \gamma_S, \alpha_S, \mathcal{M}_S, \mathcal{V}_S \rangle = \langle (r_1^\gamma, \ldots, r_n^\gamma), (r_1^\alpha, \ldots, r_m^\alpha), \mathcal{M}_S, \mathcal{V}_S \rangle$$

The two ordered rule lists γ_S and α_S define how to refine stimuli, and how to abstract from system responses. The set of named variables $\mathcal{V}_S$ contains values associated with names. $\mathcal{V}_S$ may be modified during the application of a rule from γ_S or α_S. The set of named mappings $\mathcal{M}_S$ associates concrete and abstract values. $\mathcal{M}_S$ contains grouped pairs of ordered lists, where one list contains the concrete values and the other list contains the corresponding abstract values.

A rule $r_i(m, s)$ is a binary function that takes the abstract or the concrete message as first parameter m and the current intermediate result from previous applied rules as second parameter s. r_i returns the transformed message.

Currently we use nine different types of rules that are listed in Table 1. The formal definition of the rules uses following notation: $\phi(regex, text)$ is a boolean predicate which returns true if the regular expression $regex$ matches in $text$. i denotes the start-index of the match, while ℓ denotes the end-index of the match. $\cdot$ stands for the concatenation function and $\diamond$ represents the end position of an element. $e[j : k]$ denotes the part from index j to index k of an element e. $\mathcal{V}_S(var)$ gives the value of a variable var, while $\mathcal{V}_S(var) := x$ assigns the value x to variable var. $\mathcal{M}_S(y).abs(C, i)$ denotes the retrieval of the i-th abstract value for the mapping y given the list of concrete values C. $\mathcal{M}_S(y).con(A, j)$ gives the j-th concrete value for the list of abstract values A of mapping y.

4   Empirical Results

We implemented the abstract test execution presented in Section 3 in a tool called TestExecutor and executed test cases against a commercial and an open source implementation

rule	description	formal definition
app.	Append the result of the list of rules R to s.	$app^R(m,s) = s \cdot R(m,s)$
sub-sti-tute	Given a regular expression c, this rule replaces the matched text within s by the result of R.	$sub_c^R(m,s) = \begin{cases} s[0:i] \cdot R(m,s) \cdot sub_{c,t}^R(m,s[\ell:\diamond]) & \text{if } \phi(c,s) \\ s & \text{otherwise} \end{cases}$
con-di-tion	Apply a list of rules R if the regular expression c matches within the element m.	$if_c^R(m,s) = \begin{cases} R(m,s) & \text{if } \phi(c,m) \\ s & \text{otherwise} \end{cases}$
loop	Apply a list of rules R for every match of a regular expression c in the m.	$for_c^R(m,s) = \begin{cases} for_c^R(m[\ell:\diamond], R(m,s)) & \text{if } \phi(c,m) \\ s & \text{otherwise} \end{cases}$
save	Save the result of R at location x.	$sav_x^R(m,s) = \mathcal{V}_S(x) := R(m,s)$
load	Retrieve the value stored in x.	$val_x(m,s) = \mathcal{V}_S(x)$
map-ping	Map abstract values to a concrete value and vice versa.	$con_{y,j}^R(m,s) = \mathcal{M}_S(y).con(R(m,s),j)$ $abs_{x,i}^R(m,s) = \mathcal{M}_S(x).abs(R(m,s),i)$
part	Extracts a matching regular (sub) expression r from m	$part_r(m,s) = \begin{cases} m[i:\ell] & \text{if } \phi(r,m) \\ empty\ element & \text{otherwise} \end{cases}$
text	Returns the fixed text t	$text_t(m,s) = t$

Table 1: Rules for the specification of abstraction and refinement functions.

of a Session Initiation Protocol (SIP) Registrar. Currently our TestExecutor uses UDP for the communication with the IUT. However, our approach is not limited to UDP.

SIP handles the signaling part of communication sessions between two end points. SIP defines various entities that are used within a SIP network. One of these entities is the so called Registrar, which is responsible for maintaining location information of users. SIP is a text based protocol that uses a request/response transaction model. A message consists of a start-line, indicating the message type, a message-header and a message-body. The message-header contains information like the originator, the recipient, and the content-type of the message. Message bodies of REGISTER requests are usually empty.

In order to ensure a particular system state of the IUT for each test case we reset the IUT before running a certain test case. We overcome the problems of concurrency and asynchronous communication by using the *reasonable environment assumption* [FJJV97], which says that the environments waits until stabilization after sending a single message. This means that the test execution environment waits for all responses from the IUT before sending a new message. Due to our specification's structure this assumption solves the problem of asynchronous communication as well.

For our SIP Registrar specification [APWWa, APWWb], the rule sets for γ and α comprise 66 (12 app^R, 9 sub_c^R, 7 if_c^R, 1 for_c^R, 5 val_x, 11 $con_{y,j}^R$, 9 $part_r$, and 12 $text_t$) and 36 (10 app^R, 5 if_c^R, 4 for_c^R, 3 sav_x^R, 1 $abs_{x,i}^R$, 3 $part_r$, and 10 $text_t$) rules, respectively. We use 4 mappings and 3 variables.

Table 2 outlines the results of executing test suites (1st column) containing abstract test cases against the commercial and the OpenSER Registrar in terms of the number of exe-

cuted (2nd column), passed (3rd and 8th column), failed test cases (4th and 9th column) and the overall time needed (sec.) to execute the test cases (7th and 12th column). Additionally this table shows the amount of time needed to reset the implementations (6th and 11th column), which is approximately 4,4 sec. per test case for the commercial and approximately 3,0 sec per test case for the open source implementation. The test execution with the algorithm illustrated in Figure 1 (5th and 10th column) takes approximately 7,3 sec. per test case for the commercial implementation and 3.7 sec. per test case for OpenSER. This difference is mainly caused by timeouts, since the commercial implementation sometimes does not respond to stimuli (e.g., test suite *not found*). Thus, we need to wait for the expiration of a timer (3 sec) during test execution.

Test	no.	commercial					OpenSER				
suite	tc	pass	fail	exc.	reset	sum	pass	fail	exc.	reset	sum
not found	880	0	880	6881	5607	12488	880	0	2838	2407	5245
inv. requ.	1328	0	1328	10614	5761	16375	1008	320	4501	3759	8260
unauth.	432	260	172	2714	1888	4602	130	302	1452	2409	3861
reg. ok	1488	1104	384	9143	5102	14245	1104	384	6049	3941	9990
delete	1280	16	1264	10188	5391	15579	1148	132	5201	3766	8967
Total	5408	1380	4028	39540	23749	63289	4270	1138	20040	16282	36323

Table 2: Test execution results.

Due to our specification's structure, we obtain various similar test cases. Summarized, we found 9 faults for the commercial Registrar and 4 faults for OpenSER. A detailed discussion of the detected faults can be found in [APWWa]. However, our rule based test execution framework proves to be applicable for testing this two SIP Registrar implementations.

5   Conclusion

This article presents a rule-based test execution framework that allows the execution of abstract test cases. The presented test execution framework supports the employment of model based testing in industrial sized projects. Our approach removes the requirement for writing particular test drivers for the execution of test cases. Instead we propose to use rules for the specification of the abstraction and the refinement function.

In difference to other test execution languages, like for example the standardized Testing and Test control notation (TTCN) [ETS04], we do not need to convert the complete test case into an executable format. Only required parts are considered during test execution. TTCN-3 uses more powerful programming language related constructs for test execution and test data handling. However, many of these constructs are not needed when executing (abstract) test cases derived by model based testing techniques.

References

[APWWa]   Bernhard K. Aichernig, Bernhard Peischl, Martin Weiglhofer, and Franz Wotawa. Protocol Conformance Testing a SIP Registrar: An Industrial Application of Formal Methods. In *SEFM'07*. To appear.

[APWWb]   Bernhard K. Aichernig, Bernhard Peischl, Martin Weiglhofer, and Franz Wotawa. Test Purpose Generation in an Industrial Application. In *A-MOST'07*. To appear.

[ETS04]   ETSI. Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language, 2004. ES 201 873-1.

[FJJV97] Jean-Claude Fernandez, Claude Jard, Thierry Jeron, and Cesar Viho. An Experiment in Automatic Generation of Test Suites for Protocols with Verification Technology. *Science of Computer Programming*, 29(1-2):123–146, 1997.

[JJ05]   Claude Jard and Thierry Jéron. TGV: theory, principles and algorithms. *Int. Journal on Software Tools for Technology Transfer (STTT)*, 7(4):297–315, August 2005.

[Tre96]   Jan Tretmans. Test Generation with Inputs, Outputs and Repetitive Quiescence. *Software - Concepts and Tools*, 17(3):103–120, 1996.