Electronic Communications of the EASST
Volume 47 (2012)

Proceedings of the 11th International Workshop on Graph Transformation and Visual Modeling Techniques (GTVMT 2012)

Visual Contracts as Test Oracle in AGG 2.0

Tamim Ahmed Khan, Olga Runge and Reiko Heckel

13 pages

Guest Editors: Andrew Fish, Leen Lambers
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Visual Contracts as Test Oracle in AGG 2.0

Tamim Ahmed Khan[1], **Olga Runge**[2] **and Reiko Heckel**[3]

[1] tak12@mcs.le.ac.uk
Department of Computer Sciences, Leicester University, UK

[2] o.runge@mailbox.tu-berlin.de
Department of Software Engineering and Theoretical Computer Science, TU-Berlin, Germany

[3] reiko@mcs.le.ac.uk
Department of Computer Sciences, Leicester University, UK

Abstract: A test oracle predicts expected outcomes for a set of test cases, often based on a formal, executable specification. Visual contracts are graph transformation rules describing pre- and post-conditions of a service's operations. To obtain an oracle based on visual contracts, we use the Attributed Graph Grammar System (AGG) to execute the rules, creating a simulation of the behaviour expected of the system under test.

The paper discusses the basic idea, illustrates it by an example, describes the challenges and solutions of its implementation and draws conclusions for the use of graph transformation and AGG in test oracles.

Keywords: graph transformation, services, visual contracts, test oracles

1 Introduction

Software testing relies on test oracles to predict expected test results [BY01]. In the majority of projects, oracles are implemented manually relying on testers' understanding of functional requirements to decide the correct response of the system on every given test case. As a result, they are costly in creation and maintenance and their quality depends on the correct interpretation of the requirements. Alternatively, if suitable specifications are available, oracles can be generated automatically at lower cost and with better quality [BY01].

Services hide their implementations, causing challenges to testing because established code-based techniques are no longer applicable [CP06]. Visual contracts [LSE05, LMH07] specify services by pre- and post-conditions based on an interpretation in terms of typed attributed graph transformation rules. In contrast to logic-based or algebraic specifications used in testing [Zhu03], visual contracts are easy to understand for developers familiar with UML notation or similar software modelling languages while retaining a mathematical interpretation. They are also directly executable and therefore suitable for generating automated oracles. However, the gap in abstraction between service implementation and visual models pose a number of challenges in implementing this basic idea.

Graph transformation rules can be executed using AGG [AGG07], either through its graphical user interface or via an API. Using AGG to execute our model, we provide an adapter to present

the model's functionality in a way that is comparable to the interface of the services to be tested. The test driver shown in Figure 1 implements a three step process. First, it invokes the oracle, obtaining the expected result of the test. Then, the corresponding operation of the system under test (SUT) is executed. The third step is to compare this response with the expected one and record any deviations.

The adapter is required to translate invocations of services under test into rule applications, passing and converting parameters and interpreting replies. Conversion is required because model and implementation signatures may differ, with the implementation requiring extra parameters, proving additional results, or using different types, especially for collections. Since the model is only concerned with functional aspects, we also have to filter out technical failures of the implementation, such as problems with the server or transport layer, distinguishing them from logical failures corresponding to non-applicable rules due to violation of preconditions. However, with visual contracts providing a partial specification only, even the functional aspect may be underspecified. As a further challenge, different web service implementations may report success and failure differently. Adapter and test driver need to be flexible enough to accommodate different styles of error handling and reporting, allowing for a degree of customisation.

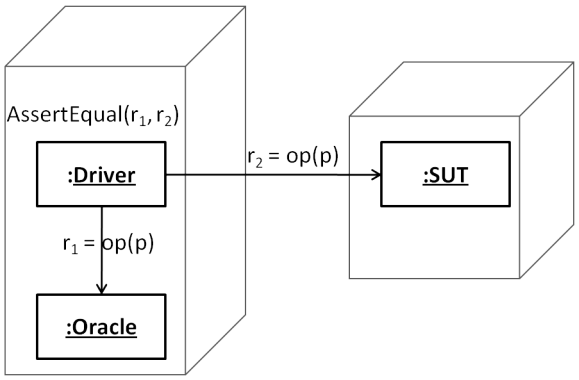


Figure 1: Oracle implementation schematic diagram

In summary, the contribution of this paper is not in generating test cases, but in helping the tester to implement them by automating the decision, if the response from the operation being tested is correct. This information is present in visual contracts and should be reused rather than reimplemented. Other test-related activities, such as debugging, are not directly affected. Therefore, the contribution is potentially relevant to all developers implementing client-side tests for services, independently of specific testing or debugging tools, as long as the tests themselves are implemented manually. While adopting visual contracts for testing is not new, their use as oracles and specifically the analysis of the challenges involved in relating executable models and implementations are original.

The rest of the paper is organized as follows. Section 2 introduces the model and the implementation of the case study. Section 3 gives a critical account of challenges we encountered in using oracles for web service testing. Section 4 introduces our treatment to the problem by means of an adapter to AGG 2.0 and custom assertions. Section 5 discusses related work before we present conclusion and outlook in Section 6.

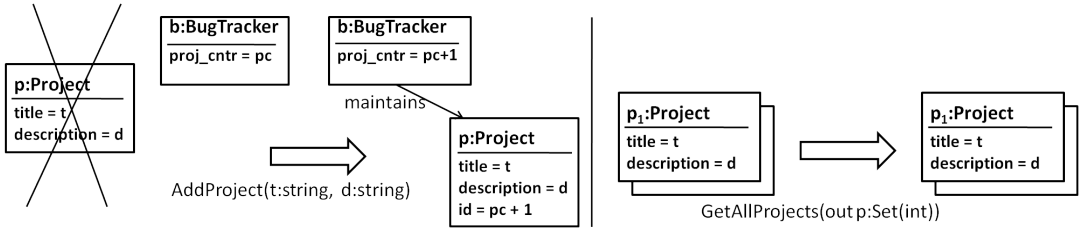
2 Visual Contracts for Services: A Case Study

As a running example, we use the case study of a Bug Tracker service derived from a desktop application[1] implemented in C#. It provides operations to manage projects and users, report faults and issues. Development teams can access fault reports and update their status. Such a service is useful for reporting and managing bug reports remotely and in an automated way. The signatures for some of the operations are shown in Listing 1. The complete interface contains 34 operations. Throughout the rest of the paper, we speak of the web service as the *implementation* and refer to the visual contracts as the *model*.

Listing 1: BTSImplementationInt

```
        ...
namespace BTSysWebService {
        ...
    public class BTSysService : System.Web.Services.WebService   {
        ...
        [WebMethod]
        public String AddProject(String title, String description) { ...  }
        [WebMethod]
        public String AssignProject(Int32 pId, Int32 uId)  { ... }
        [WebMethod]
        public String UpdateProject(Int32 pId, String title, String description){...}
        [WebMethod]
        public List<ProjectInfo> GetAllProjects() { ...  }
        ...
```

We specify operations by visual contracts. A contract allows us to describe what must hold prior to invocation of an operation and what should be true once the invocation is complete. A *visual contract*, as shown in Figure 2, is a specification of pre- and post-conditions by a pair of graph patterns representing the left- and right-hand sides of a rule [LSE05, LMH07].



(a) visual contract: addProject (b) visual contract: getAllProjects

Figure 2: Visual Contracts (a) and (b)

We associate a signature with each visual contract distinguishing formal input and output parameters. Parameters are variables occurring in attribute expressions. Consider Figure 2(a), where the signature $addProject(t : String,\ d : String)$ has parameters t and d which are also used

[1] available at http://btsys.sourceforge.net/

in the contract *addProject* to represent possible values for *title* and *description*. The signature associated with the visual contract in Figure 2(b) has a set of integers as output. The interface listing these signatures is shown in Figure 3.

The pre-condition of the visual contract shown in Figure 2(b) allows us to find a match for any number of *Project* nodes. As seen from the identical post-condition, the rule has no effect. This allows us to model a query on the system state. The model is formally represented as a typed attributed graph transformation system (TAGTS) where visual contracts are specified as rules over the type graph representing the class diagram of the service as shown in Figure 5.

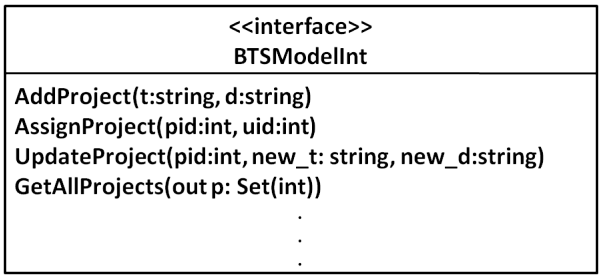


Figure 3: Model level signatures

The model specifies functional requirements only, without addressing error handling or reporting. This may result in differences between model and implementation signatures. For example, at implementation level *addProject* has return type *String* to carry success or error messages of the operation, while the model only records applicability or otherwise of the corresponding rule.

3 Challenges

In our approach, the oracle executing the TAGTS model is invoked through an adapter to utilize the API of AGG 2.0 as shown in Figure 4. However, the comparison of the expected with the actual result raises a number of challenges, which we will discuss below.

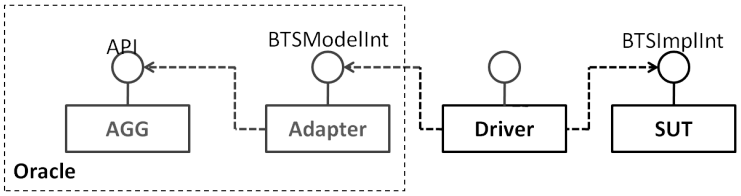


Figure 4: Component diagram

3.1 Model as Oracle

We represent visual contracts as rules using AGG. The tool does not provide a mechanism to associate a rule with a signature. The driver, however, requires to invoke the model by the same inputs used to invoke the implementation and needs to receive the computed outputs to compare the predicted and actual results. This is only possible if we can provide a mapping between the

inputs and outputs of the model signature to the rule's inputs and outputs, which is the role of the adapter in Figure 4.

3.2 Partiality of Visual Contracts

A visual contract may specify the intended behaviour of the implementation only partially. Therefore, if the oracle predicts success and the implementation reports logical failure, this could either be due to an error in the implementation or the underspecified precondition of the contract. While comparing model and implementation responses, such a case should not be reported as a failed test, but as a warning, providing sufficient details so that the developer can decide the correct interpretation. Table 1 details all possible cases and will be more fully explained in the next subsection. The 5th row represents the situation where the oracle gives a response while the implementation reports a logical failure.

An underspecified postcondition would lead to a lack of synchronisation between model and implementation state, because changes performed on the latter would not have to be matched by the former. Therefore, postconditions are assumed to be complete.

3.3 Failure Handling

There are different ways in which failures can be reported to the test driver or client. We distinguish them by raising the following questions.

1. What is the origin of the failure?
2. How was the failure presented?
3. How is the failure interpreted?

A failure may have its origin either on the server or in communication. Server-side failure can be due to logical or technical reasons. Logical failures occur if the pre-condition of an operation is not satisfied, i.e., the application is invoked, but not executed correctly or not at all. Technical failures can be down to a variety of reasons, such as the database being off-line, server-side system failures, power fluctuations, hardware issues, etc. Communication failures result from loss of network access, congestion causing delays, etc.

A server-side failure presents itself as an exception, a fault message, or an application-specific error code, while a communication failure shows as an exception (timeout) on the client side.

We interpret these failure presentations as follows. If the client-side receives a logical failure, we expect a violation of the contract's pre-condition. If the client receives a technical failure, the oracle cannot provide a matching response since the model only covers the functional aspect. Similarly, if the client receives a communication failure, the comparison between expected and actual response is meaningless.

We list the cases in Table 1, where r indicates equal responses from oracle and implementation in case of successful execution and r_1, r_2 represent successful responses that differ from each other. Equality "=" shows that the test was executed and successful whereas "undefined" means that, for all we know, the test case was not executed. If responses do not match due to non-applicability of a pre-condition in the model, this is represented by "$\neq_{pre}$", whereas if model and implementation produce different responses, this is represented by "$\neq_{post}$". The first and the third

Oracle response	Result	*SUT* response
r	$=$	r
r_1	$\neq_{post}$	r_2
non-applicable	$=$	logical-failure
non-applicable	undefined	technical failure
r	?	logical failure
r	undefined	technical failure
non-applicable	$\neq_{pre}$	r

Table 1: Exception response table

rows represent cases where both responses match, i.e., both show either successful execution or failure. The second row represents cases where the pre-condition was satisfied, yet the output was different. The forth and the sixth rows represent cases where the SUT experienced a technical failure and hence the test case was not executed. The fifth row represents cases where the oracle generated a response and the SUT returned a logical error. This case is reported to the developer as a possible failure that needs further investigation, as it may be due to a faulty implementation or a partially specified visual contracts. This is marked with "?" in the results column. Lastly, the seventh row represents cases where the oracle reported non-satisfaction of pre-condition, whereas the implementation generated a response.

3.4 Adaptation of Output Types

If implementation and model share the same signature, expected and actual output can be compared directly. However, there are cases where the implementation returns a result in the form of a complex type, such as a collection of objects. In such cases, the oracle returns its response as a set of nodes and the adapter needs to process this set into a suitable form for the driver to carry out the comparison.

The implementation signatures can extend the model signatures by providing an additional response to indicate if the operation was successful. This response can be in the form of a numerical error code, a string, or a Boolean. One example from our case study is $AddProject(\ldots)$, where the success message of the implementation is a string "Data saved successfully...". The model instead reports successful execution of a rule by means of a Boolean return via the AGG API. We therefore not only need to maintain separate signatures, but also adapt the results to make them comparable.

4 Using AGG as Oracle

Having listed the challenges in using visual contracts as oracles, we discuss how we have used the AGG 2.0 [2] engine [AGG07] and appropriate adapter and driver implementations to overcome these challenges.

[2] available at http://user.cs.tu-berlin.de/~gragra/agg/

4.1 Model as Oracle

In order to link model signatures with production rules in AGG, we map their formal parameters to the parameters and variables in the attribute context of the rules. An example is shown in Figure 5.

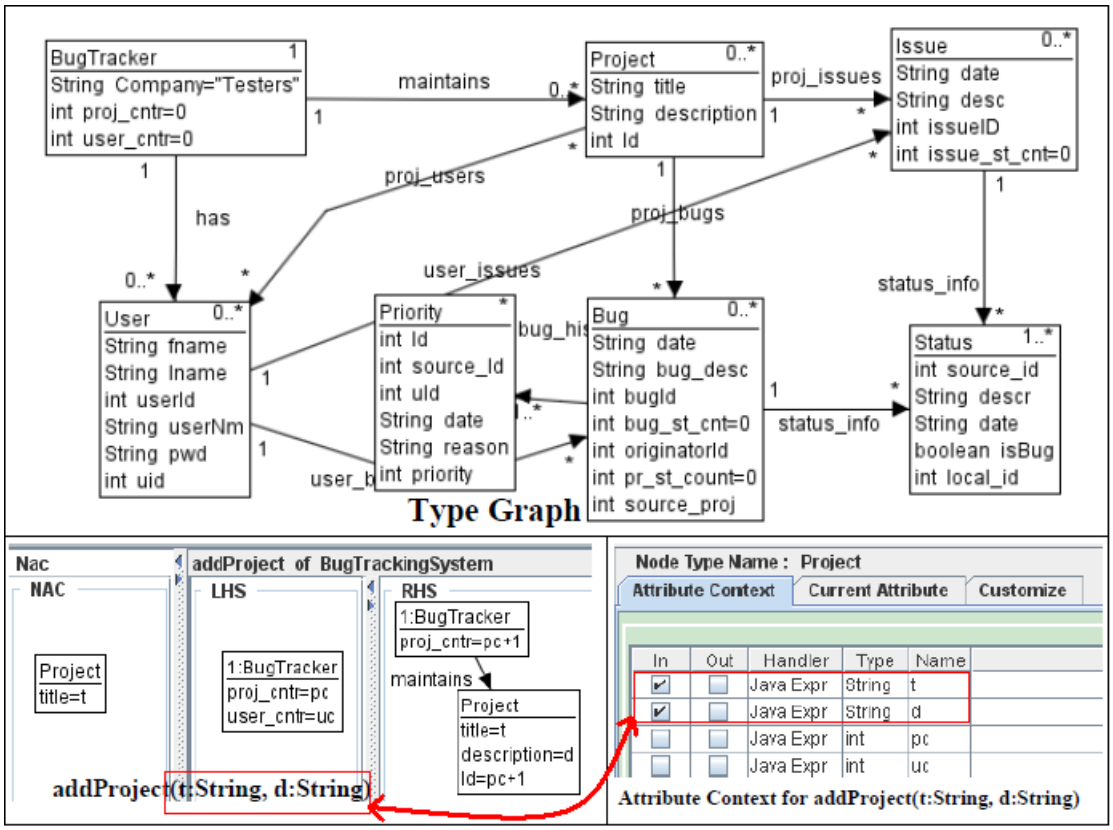


Figure 5: Rules and typed graph example

This enables the driver to use the same input parameters to invoke the model and the implementation operations. This is enabled by the adapter, which wraps an invocation of the AGG API inside a call to an operation based on the model signature. Examples of such invocations are shown in Listing 3 (e.g., on line 12).

However, if the signatures involve multi-objects as shown in Figure 2(b), we make use of rule schemes where an amalgamated transformation [GBEE11] returns the set of the nodes corresponding to the multi-object on the right-hand side of the rule. This is shown in Figure 6 where the start graph contains two projects and the right-hand side of the amalgamation shows both of them selected as nodes. Our rule schemes implement all-quantified operations on recurring graph patterns. The kernel rule is a common subrule of a set of multi-rules. It is matched only once, while multi-rules are matched as often as suitable matches are found. In AGG an amalgamated rule is constructed from all matches found for multi-rules that share the match of the kernel rule.

In the example of Figure 6, the kernel rule of the rule scheme for operation $getAllProjects(\ldots)$

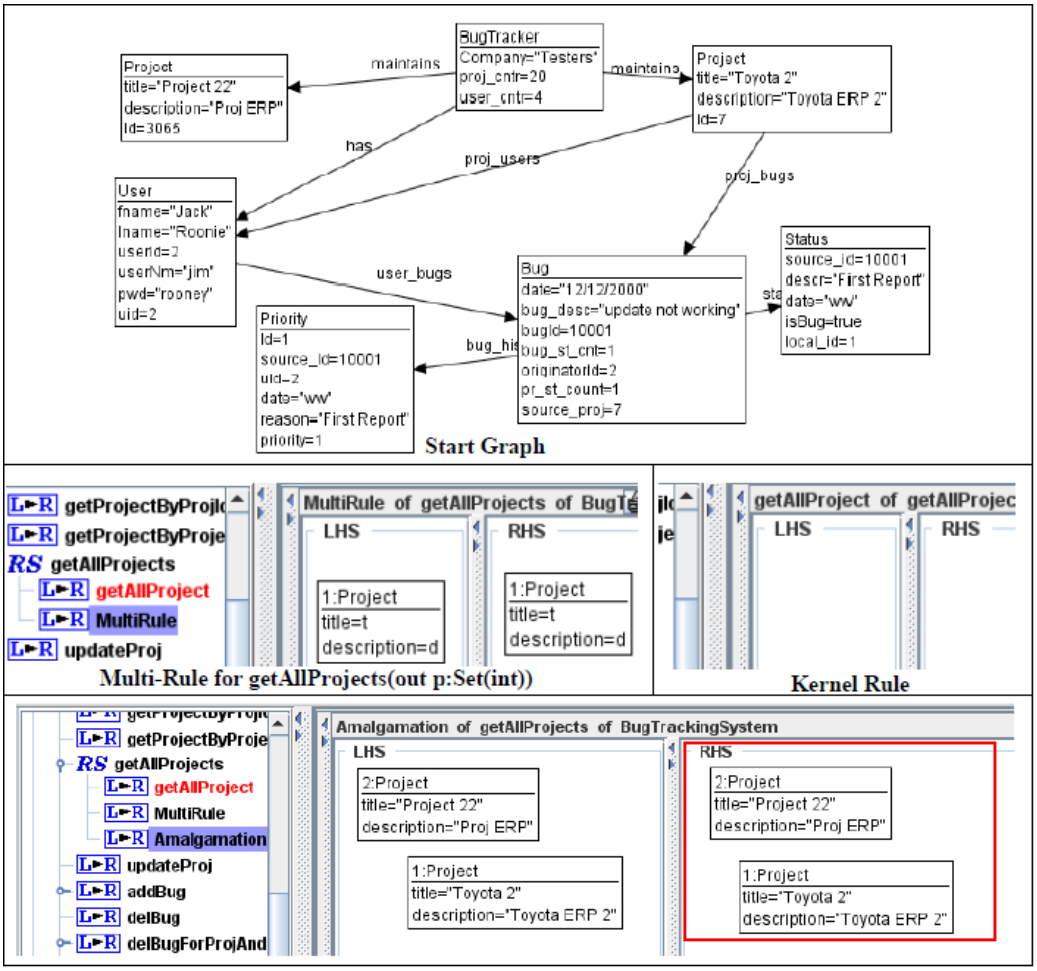


Figure 6: Rules and typed graph example

is the empty rule. That means the kernel rule (and so the rule scheme) is applicable to each graph, including the empty graph. In AGG, the multi-rule of the rule scheme *getAllProjects* identifies a project. When the multi-rule is matched to all projects, we get as many project nodes in the left- and right-hand side of the amalgamated rule. Our adapter provides access to the list of these nodes and processes them to return the data for comparison to the driver.

At the start of testing, we assume that the start graph of the model and the initial state of the implementation are in sync. Typically, neither of them have any non-mandatory data. The synchronisation of model and implementation states is maintained by their coevolution as long as the results match, assuming that post-conditions of visual contracts are completely specified.

4.2 Partiality of Visual Contracts

It remains to deal with underspecified preconditions. The adapter allows to observe applicability of rules and their generated output. If the intended behaviour is specified only partially, the model

may generate a response while the implementation returns a logical failure. In this case, we provide developers with both responses and the stack trace detailing the reasons for the mismatch. The developers can choose to ignore the results and subsequent test executions using annotation *@Ignore* if a detailed analysis reveals that the response is due to a partially specified visual contract.

Based on the information in a log file, we can also compile a summary report where all detailed test reports are condensed as shown in Listing 2, referring to the test cases in Listing 3. The first row represents a successful execution of model and implementation. The second row represents logical failure responses from model and implementation. Hence, the system passes both tests.

Listing 2: JUnit Test:AddProject

SN	Test Case	Oracle	SUT	Result
==	=========	======	===	======
1	AddProject(”title”,”description”)	applicable	Data saved successfully...	=
2	AddProject(”title”,”description”)	not−applicable	Error occurred while saving data...	=
...				

4.3 Failure Handling

Our discussion in subsection 3.3 revealed that comparison between the responses of oracle and the implementation is possible only in a subset of cases. In case of a technical server or communication failure, there is either no response or a timeout on the implementation side. Since the oracle only covers functional aspects, in this case comparison is not possible. In order to avoid further processing, we check that there is no timeout and use an assert statement $assertNotNull(\ldots)$ before comparing the oracle's with the implementation's output, as shown in Listing 3 (line 7 and line 22).

If the client receives a logical failure from the server, we check the response of the oracle. If the oracle returns $true$, our custom assertion $assertBothSucceededOrBothFailed(\ldots)$ evaluates to $false$ since there is a mismatch between the two responses. However, if the oracle also reports a logical failure, our custom assertion evaluates to $true$. Listing 3, line 25, demonstrates the usage of the assertion. We have already added a project (line 5 and line 12) and are trying to insert it again (line 20 and line 23) with the same credentials. Since the oracle's response is $false$ and the implementation returns $true$, our custom assertion returns $false$.

It is important to point out that we do not perform literal comparison between the expected and the actual result since the implementation can report a failure in a variety of ways. Examples include error messages, e.g., ”error occurred...” or ”Error code: 9876, check documentation”, etc. Therefore, our custom assertions are general enough to be able to compare a variety of different failure representations coming from the implementation with the oracle's response. The composition of $assertBothSucceededOrBothFailed(\ldots)$ is such that it has two parts, $assertBothSucceeded(\ldots)$ to compare the successful cases and $assertBothFailed(\ldots)$ to compare the failure cases. The second part deals with cases where the oracle gives a Boolean response. We allow the developer to define how the implementation reports failure, based on a set of predefined alternatives.

Listing 3: JUnit Test:AddProject

```
...

1.    org.tempuri.BTSysServiceStub.AddProject addProject160 = ...;
2.    addProject160.setTitle(title);
3.    addProject160.setDescription(description);
4.    myAssert.successMessage = result;
5.    AddProjectResponse resp = stub.AddProject(addProject160);
6.    assertNotNull(resp);
7.    String res1 = resp.getAddProjectResult();
8.    aggEngine agg = new aggEngine();
9.    ArrayList<String> list = new ArrayList<String>();
10.   list.add("\"" + title + "\"");
11.   list.add("\"" + description + "\"");
12.   boolean res2 = agg.aggResult("C:\\localapp\\bts.ggx", "addProject", list);
13.   assertNotNull(res2);
14.   assertTrue(res2);
15.   myAssert.assertBothSucceededOrBothFailed((Object) res2, (Object) res1);
16.   agg.save();

      ...
17.  org.tempuri.BTSysServiceStub.AddProject addProject161 = ...;
18.  addProject161.setTitle(title);
19.  addProject161.setDescription(description);
20.  AddProjectResponse resp2 = stub2.AddProject(addProject161);
21.  assertNotNull(resp2);
22.  String res3 = resp2.getAddProjectResult();
23.  boolean res4 = agg.applyRule("addProject", list);
24.  assertNotNull(res4);
25.  assertFalse(res4);
26.  myAssert.assertBothSucceededOrBothFailed((Object) res4, (Object) res3);
      ...
```

We are left with cases where both implementation and oracle have reported successful invocation. The next subsection discusses the different cases for comparing output values.

4.4 Adaptation of Output Types

If the signatures of model and implementation are the same, the output value is passed to the driver through the variable *result*. The standard $assertEquals(\ldots)$ can deal with this case.

If the implementation signature is an extension of the model signature, we proceed as follows. The oracle informs the driver if the pre-condition was satisfied. The custom assertions allow us to compare the oracle's response with the implementation's, recording the latter. This allows the driver to know how a particular implementation responds in success cases. This is demonstrated in Listing 3 where we first check, by using *assertNotNull*, if the service invocation was successful and by using *assertTrue* if the rule application was successful. The driver then uses a custom assertion $assertBothSucceededOrBothFailed(\ldots)$ by initialising the expected result string to see if the results were compatible (lines 4 to 15).

If the implementation returns execution results in terms of a complex type, we access the data in the response object and the resulting set of nodes from a collection named *nodeStruct* in the oracle and use custom assertion $assertSetEquivalent(\ldots)$ to compare the two sets of values. Listing 4 presents an example. We receive the multi-object from the implementation (Listing 4, line 1) and extract the result in the form of a list. We invoke our oracle and access the result as

another list (Listing 4, line 6) and test by using our custom assertion, *assertSetEquivalent*(...) (line 8), if the two responses were the same.

Listing 4: JUnit Test:GetAllProjects

```
   ...
1.  ProjectInfo[] result = resp.getGetAllProjectsResult().getProjectInfo();
2.  List<List<String>> projectList = new ArrayList<List<String>>();
3.  for (int i = 0; i < result.length; i++) { ... }

    ...
4.  boolean res2 = agg.aggEngineGetAll("C:\\localapp\\bts.ggx", "getAllProjects");
5.  List<List<String>> projects = new ArrayList<List<String>>();
6.  projects = agg.nodeStruct;
7.  assertNotNull(res2); assertTrue(res2);
8.  myAssert.assertSetEquivalent(projects, projectList);
   ...
```

5 Related Work

We first discuss different approaches to oracle development, based on different types of artifacts, and then address the use of visual contracts in the context of testing.

Approaches to automate test oracles consider system specifications [RAO92], documentation [PMD+98], and parallel implementation [Zhu03, BY01]. Test oracles in [TCP+05] for web services require several implementations for each service and a training phase to generate the oracle. Another approach by [CCL05] is based on the idea of metamorphic testing, which emphasizes the usage of relations between inputs and outputs of several executions of a method under scrutiny. This allows the reuse of existing test cases to generate more test cases. Test oracles for web services are also proposed in [ABFJ08] where tables describing sets of test cases, called test sheets, are used. These tables contain inputs and sets of possible outputs. The approach builds on concepts defined in the Framework for Integrated Test[3], suggesting a manual process for the creation of test oracles for web services. Our approach differs from the proposals cited above since we propose a mechanism to execute visual service specifications and do neither rely on training, nor require the additional overhead of an alternative implementationion as in metamorphic testing.

Visual contracts have been used for testing in [HL04, GMWE09]. [HL04] have specified visual contracts as production rules in a TAGTS for deriving functional tests from contracts, using JTest and Parasoft for implementation. Visual contracts have been used to formalize pre- and post-conditions of use cases and as test models to generate logical test cases in [GMWE09]. The paper establishes a relation between contracts and UML specifications as well as introduces test suite generation for required and provided interfaces.

We focus on oracles rather than test case generation, which is a different questions, but use the same kind of specification. Our work provides an implementation using AGG and JUnit and provides a means of executing the specifications through an adapter linking the model signatures with production rules in AGG.

[3] available at http://fit.c2.com

6 Conclusion and Outlook

We have used high-level visual contracts for oracle development, considering the challenges arising from the abstraction gap between model and implementation. Assuming availability of the visual contracts, their use as oracles comes at little extra cost while reducing the possibility of errors in the manual implementation of tests and the effort for maintaining and evolving tests.

However, there are some weaknesses in the current implementation. Our oracle consists of models executed in AGG, which does not directly support the association of rule signatures to productions rules. We identify formal parameters with variables in the rules' attribute context, but this is a manual process. An automated solution would eliminate the possibility of a mismatch between the rule and the signature. Another limitation is that we can only use basic data types as input parameters, while outputs can be collection types.

In order to make the approach applicable in practice, better support is required to bridge the gap between model and implementation. That means for the adapter implementing the model interface to be generated automatically. Adopting the techniques described in Section 4.1, this is straightforward except for contracts containing multi-objects which are realized in AGG by means of amalgamated rules and for which the extraction of input and output parameters has to be implemented manually. More generally, this would require integrating into AGG and its API the concept of rule signatures defined over the attribute context of a rule. The test driver on the other hand, with its customizable assertions, is required to accommodate variations in the use of exceptions and implementation-specific error messages, which can vary widely based on the conventions of the application at hand. Automation here is meaningful only if these conventions could be formalized and made part of the model. We plan to extend our work by providing test coverage based on the criteria set out in [HKM11].

Bibliography

[ABFJ08] C. Atkinson, D. Brenner, G. Falcone, M. Juhasz. Specifying High-Assurance Services. *Computer* 41(8):64 –71, aug. 2008.

[AGG07] AGG. AGG - Attributed Graph Grammar System Environment. http://tfs.cs.tu-berlin.de/agg, 2007.

[BY01] L. Baresi, M. Young. Test oracles. *University of Oregon, Dept. of Computer Science*, 2001.

[CCL05] W. Chan, S. Cheung, K. Leung. Towards a metamorphic testing methodology for service-oriented software applications. In *Quality Software, 2005. (QSIC 2005). Fifth International Conference on*. Pp. 470 – 476. sept. 2005.

[CP06] G. Canfora, M. D. Penta. Testing Services and Service-Centric Systems: Challenges and Opportunities. *IT Professional* 8:10–17, 2006.

[GBEE11] U. Golas, E. Biermann, H. Ehrig, C. Ermel. A Visual Interpreter Semantics for Statecharts Based on Amalgamated Graph Transformation. *ECEASST* 41, 2011.

[GMWE09] B. Güldali, M. Mlynarski, A. Wübbeke, G. Engels. Model-Based System Testing Using Visual Contracts. In *Proceedings of Euromicro SEAA Conference 2009, Special Session on "Model Driven Engineering"*. Pp. 121–124. IEEE Computer Society, Washington, DC, USA, 2009.

[HKM11] R. Heckel, T. A. Khan, R. Machado. Towards Test Coverage Criteria for Visual Contracts. *Electronic Communications of the EASST* 41, 2011.

[HL04] R. Heckel, M. Lohmann. Towards Contract-based Testing of Web Services. *Electronic Notes in Theoretical Computer Science* 82:2003, 2004.

[LMH07] M. Lohmann, L. Mariani, R. Heckel. A Model-Driven Approach to Discovery, Testing and Monitoring of Web Services. *Test and Analysis of Web Services*, pp. 173–204, 2007.

[LSE05] M. Lohmann, S. Sauer, G. Engels. Executable Visual Contracts. In *VLHCC '05: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*. Pp. 63–70. IEEE Computer Society, Washington, DC, USA, 2005.

[PMD+98] D. Peters, S. Member, I. David, L. Parnas, S. Member. Using Test Oracles Generated from Program Documentation. *IEEE Transactions on Software Engineering* 24:161–173, 1998.

[RAO92] D. J. Richardson, S. L. Aha, T. O. O'Malley. Specification-based test oracles for reactive systems. In *Proceedings of the 14th international conference on Software engineering*. ICSE '92, pp. 105–118. ACM, New York, NY, USA, 1992.

[TCP+05] W.-T. Tsai, Y. Chen, R. Paul, H. Huang, X. Zhou, X. Wei. Adaptive Testing, Oracle Generation, and Test Case Ranking for Web Services. In *Proceedings of the 29th Annual International Computer Software and Applications Conference - Volume 01*. COMPSAC '05, pp. 101–106. IEEE Computer Society, Washington, DC, USA, 2005.

[Zhu03] H. Zhu. A Note on Test Oracles and Semantics of Algebraic Specifications. In *Proceedings of the Third International Conference on Quality Software*. QSIC '03, pp. 91–98. IEEE Computer Society, Washington, DC, USA, 2003.