On Model-Based Regression Testing of Web-Services Using Dependency Analysis of Visual Contracts

Tamim Ahmed Khan and Reiko Heckel

Department of Computer Sciences, Leicester University, UK
{tak12, reiko}@mcs.le.ac.uk

Abstract. Regression testing verifies if systems under evolution retain their existing functionality. Based on large test sets accumulated over time, this is a costly process, especially if testing is manual or the system to be tested is remote or only available for testing during a limited period. Often, changes made to a system are local, arising from fixing bugs or specific additions or changes to the functionality. Rerunning the entire test set in such cases is wasteful. Instead, we would like to be able to identify the parts of the system that were affected by the changes and select only those test cases for rerun which test functionality that could have been affected.

This paper proposes a model-based approach to this problem, where service interfaces are described by visual contracts, i.e., pre and post conditions expressed as graph transformation rules. The analysis of conflicts and dependencies between these rules allows us to assess the impact of a change of the signature, contract, or implementation of an operation on other operations, and thus to decide which of the test cases is required for re-execution. Apart from discussing the conceptual foundations and justifications of the approach, we illustrate and evaluate it on a case study of a bug tracking service in several versions.

1 Introduction

Service-oriented systems pose new challenges to client-side testing [1]. Problems arise from the lack of access to and control over the implementation (let alone the code), which prohibit the use of white box techniques and may limit (due to the cost for using a service or the limited time available) the number of tests that can be executed [2].

Evolution in software systems is inevitable to keep them abreast with the changing needs of businesses. To assess and assure that there is no deviation of the existing functionality, regression testing uses a comprehensive set of test cases to reevaluate every new version. Such regression test suites are accumulated over time and can be large and costly to run [3]. In many cases, however, the impact of a particular evolution step is limited to a small part of the system, especially if maintenance is concerned with minor corrections or additions. In such cases it would be beneficial to select only those test cases for rerun which exercise parts of the system directly or indirectly affected by the changes.

Following the classification in [4], a test case in a regression test suite can be *obsolete* (*OB*) if it is no longer applicable to the new version, *reusable* (*RU*) if it is still applicable and *required* (*RQ*) if it tests functionality affected by the changes. Given two consecutive versions of the system $V1$ and $V2$ together with information about the changes from one to the other, the problem is therefore to classify a set of test cases executable over $V1$ into the three categories in such a way that any faults detected in $V2$ by executing RU are also detected running RQ only. In this paper we will provide a classification and argue for its correctness in the above sense both conceptually (based on a formalisation of the problem in terms of graph transformation systems) and by an evaluation through a small but non-trivial case study of a service in three versions.

Since code is not available, our treatment of the problem is based on model-based service descriptions at the level of interfaces. In this way we also abstract from details of the programming language, supporting the platform-independent nature of services. Semantic information at the interface level is expressed by means of typed graph transformation systems [5] presented as visual contracts [6]. This has the advantage of using a visual specification in line with mainstream software modelling languages such as the UML, while at the same time retaining a formal semantics and mathematical theory. Based in particular on the theory of conflicts, causality and concurrency of graph transformations we analyse data dependencies between and within test cases based on which we determine the impact of change and thus the set of required test cases.

We perform an evaluation of our method based on three versions of a Bug Tracking service adapted from an open source application in C#. Defining and classifying test cases for the three versions, we are interested in both the number of test cases saved by the classification and its continued ability to find all the faults. We use error seeding techniques to assess the quality of both the complete and reduced test sets.

The paper is organised as follows. Section 2 introduces the basic concepts of our approach, including the specification of visual contracts by graph transformation rules and the use of trace theory to provide an observational semantics appropriate for testing. Section 3 presents our evolution scenario and describes and applies our methodology. The evaluation is reported on in Section 4 while related work is discussed in Section 5 before we conclude the paper.

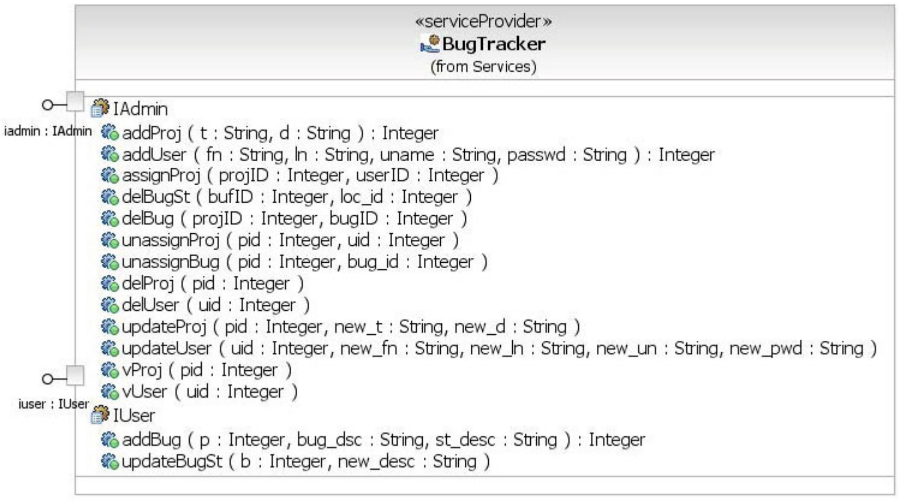
2 Visual Contracts and Trace Semantics

In model-based testing of services we are potentially concerned with three layers of abstraction: those of *implementation, interface model*, and *observable behaviour*. While the implementation is hidden, we will require information about changes, such as for which operations from the interface the implementation has been modified. The interface model details operation signatures and accompanying data types as well as describing the semantics of operations in terms of pre- and post conditions. Such descriptions may be available in diagrammatic form at design time, but also in machine-readable form at run time. The observable behaviour is expressed in terms of sequences of messages representing invocations to service operations, e.g., as part of a test case being executed. In order to define precise criteria for distinguishing different categories of test cases, we have to study the relation between interface models and observations.

Visual Contracts. We represent service interface models by typed graph transformation systems (TGTS). A TGTS $\mathcal{G} = (TG, Sig, R)$ consists of a type graph TG modelling the public data types available at the interface, a set of rule signatures Sig providing operation names with parameter declarations $p(x_1 : s_1, \ldots, x_n : s_n)$. Here $x_i : s_i$ represents a formal parameter x_i of type s_i. A set of rules R is associated to these signatures, describing the behaviour of the corresponding operations as visual contracts [7,8], i.e., pre- and post conditions $L \to R$ shown as object diagrams. We write $p(x_1 : s_1, \ldots, x_n : s_n) : L \to R \in \mathcal{G}$ if there is a rule $L \to R$ associated with an operation signature $p(x_1 : s_1, \ldots, x_n : s_n)$.

Example 1 (Bug Tracker service). In order to illustrate the use of these models we present a case study of a Bug Tracker service, to be used as a running example throughout this paper. Three consecutive versions of the service have been derived from an open source desktop application[1] by replacing its GUI by a service interface. Such a service could be useful, for example, in order to allow automatic bug reports through applications detecting faults or in order to integrate bug tracking data into higher level functions.

In its basic version, bug tracking serves the communication between developers, users, testing team, etc. Once a bug has been added by the user who discovered it, its status can be updated by the developers and testers until the issue is resolved. In addition to the interface provided to the user, we have also created an administrative interface to add, update, or delete projects and users. Both interfaces are listed in Fig. 1.



IAdmin
addProj (t : String, d : String) : Integer
addUser (fn : String, ln : String, uname : String, passwd : String) : Integer
assignProj (projID : Integer, userID : Integer)
delBugSt (bufID : Integer, loc_id : Integer)
delBug (projID : Integer, bugID : Integer)
unassignProj (pid : Integer, uid : Integer)
unassignBug (pid : Integer, bug_id : Integer)
delProj (pid : Integer)
delUser (uid : Integer)
updateProj (pid : Integer, new_t : String, new_d : String)
updateUser (uid : Integer, new_fn : String, new_ln : String, new_un : String, new_pwd : String)
vProj (pid : Integer)
vUser (uid : Integer)
IUser
addBug (p : Integer, bug_dsc : String, st_desc : String) : Integer
updateBugSt (b : Integer, new_desc : String)

Fig. 1. Bug Tracking interfaces for Admin and User

A conceptual data model for the service, limited to the data visible to its clients, is presented in Fig. 2(a). Beside a top level class BugTracker, we find User and Project data as well as Bug and Status information. A selection of rules representing visual contracts are shown in Fig. 2(b). For example the addBug rule describes how a bug

[1] Available at `http://btsys.sourceforge.net/`

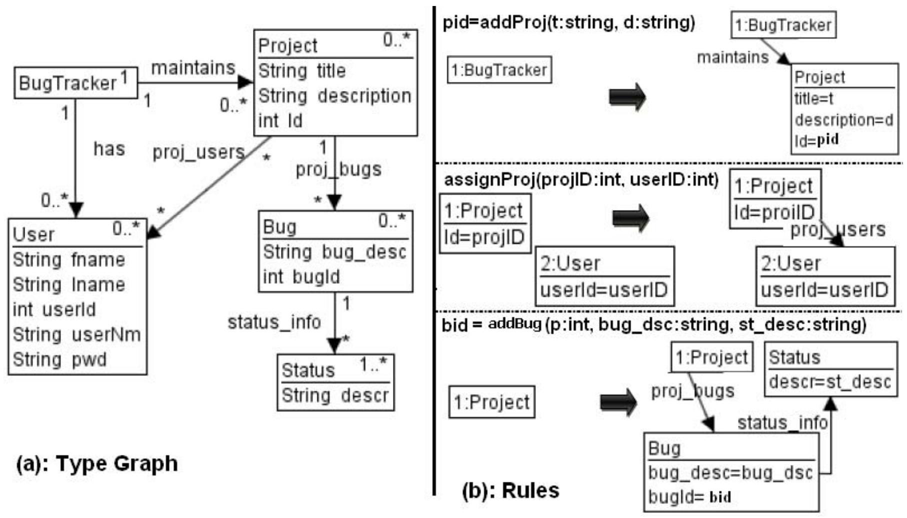


Fig. 2. Type graph and rules

report is added to the database, assuming an existing project and adding Bug and Status objects.

Observational Semantics. Graph transformation systems come with an execution semantics, i.e., we can simulate the implementation at the interface level by means of rule applications. Given a graph G representing a state of the system and an operation $p(x_1 : s_1, \ldots, x_n : s_n) : L \to R \in \mathcal{G}$, we can attempt an invocation $p(a_1, \ldots, a_n)$, substituting formal parameters x_i in $p(x_1 : s_1, \ldots, x_n : s_n)$ by actual data values a_i found in G_0. If there exists a match $m : L \to G$ embedding L into G such that $m(x_i) = a_i$, i.e., m is compatible with the instantiation of parameters, the rule can be applied resulting in a *transformation step* $G \overset{p,m}{\Longrightarrow} H$. The observation or *label* of this step, $\delta(G \overset{p,m}{\Longrightarrow} H) = p(a_1, \ldots, a_n)$ is given by the rule name with actual parameters, while the state and the actual rule remain hidden. The set of all possible observations for the (implicit) signature Sig is denoted by O whereas the set of all possible sequences provided by Kleene closure of O is denoted by O^*.

Assuming a start state represented by graph G_0, selecting rules and matches we can produce a *transformation sequence* $\rho = G_0 \overset{p_1,m_1}{\Longrightarrow} G_1 \overset{p_2,m_2}{\Longrightarrow} \cdots \overset{p_n,m_n}{\Longrightarrow} G_n$. The set of all these sequences is $\mathcal{D}er(\mathcal{G})$ and the observation function δ extends to such sequences, i.e., $\delta : \mathcal{D}er(\mathcal{G}) \to O^*$. That means, $\delta(\rho)$ is the sequence of labels obtained as observations of the steps of ρ.

Example 2 (transformation sequence and observation). Consider the bug tracking system whose interface is shown in Fig. 1 and the type graph and rules are shown in Fig. 2 with a start state having only one project and one user as represented by graph G_0 in Fig. 3. Transformation sequence $G_0 \overset{addProj,m_1}{\Longrightarrow} G_1 \overset{addUser,m_2}{\Longrightarrow} G_2 \overset{assignProj,m_3}{\Longrightarrow} G_3$ shown in Fig. 3 creates a new project and user and assigns the project to the user.

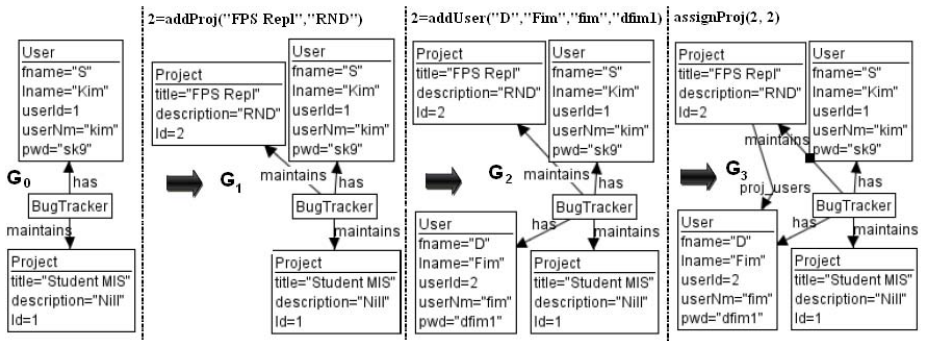


Fig. 3. Transformation sequence

The corresponding sequence of observations is $2 = addProj(“FPSRepl”, “RND”)$; $2 = addUser(“D”, “Fim”, “fim”, “dfim1”)$; $assignProj(2, 2)$ where return values are 2 in both $addProj(\ldots)$ and $addUser(\ldots)$.

In order to ensure that labels carry enough information for observations to reflect faithfully the behaviour at the interface level, we have to make a number of assumptions. First, we assume that all objects can be uniquely identified by their collection of attributes, and that these attributes are always fully defined in the states of the system. This is of course a requirement for the initial state, but also for the rules specifying the operations, which have to preserve these properties. Second, operation signatures need to carry enough parameters to identify uniquely the match of a transformation. This is satisfied, for example, if each signature lists *id* attributes for all elements of its rule's left-and right-hand side as parameters, thus specifying completely the embedding of the rule into graphs G and H. In most practical cases, however, parameters will only need to identify some anchor elements, which will then determine the other elements in the match and co-match. For example, as stated in the cardinality of 1 on the *status_info* association, a Status object will always refer to a unique Bug, so identifying the Status we implicitly know the Bug as well. These conditions are formalised in [9] in terms of morphisms of attributed graphs. If they are satisfied, the observation function δ is called *faithful*.

Conflicts and Dependencies. In order to understand if two observations can be part of the same invocation sequence, or if they can occur in that sequence in a given order, we have to analyse causal dependencies and conflicts between transformations and represent them at the level of labels. At the model level, we say that

- a competing transformation $G \overset{p_2,m_2}{\Longrightarrow} H_2$ *disables* $G \overset{p_1,m_1}{\Longrightarrow} H_1$ if the match for p_1 is destroyed by the application of p_2;
- a consecutive transformation $G_1 \overset{p_2,m_2}{\Longrightarrow} G_2$ *requires* $G_0 \overset{p_1,m_1}{\Longrightarrow} G_1$ if the application of p_1 creates elements required for the application of p_2.

These relations are essentially those of asymmetric event structures [10]. The asymmetry arises from the interplay of deletion and preservation, which is typical to all computational models distinguishing read and write access to resources.

If two steps are not in conflict or dependent, they are independent. Two independent consecutive steps can be swapped. The standard model of concurrency for graph transformation systems, based on the so called *shift-equivalence* $\equiv_{sh} \subseteq Der(\mathcal{G}) \times Der(\mathcal{G})$, abstracts from the order in which independent steps are applied, considering all derivations equivalent that represent serialisations of the same concurrent process. The quotient $Der(\mathcal{G})/\equiv_{sh}$ defines the set of concurrent derivations in the system.

In order derive a concurrent *observational* semantics, following [9] we lift the *disables* and *requires* relations to the level of labels. For two labels l_1, l_2 and transformations ρ_1 and ρ_2 such that $l_i = \delta(\rho_i)$, we write

- $l_1 \nearrow l_2$ if ρ_2 *disables* ρ_1;
- $l_1 < l_2$ if ρ_2 *requires* ρ_1;

If l_1, l_2 are unrelated by $\nearrow$ and $<$, they are independent, written $l_1 \mid l_2$.

In order to calculate conflicts and dependencies at the level of labels we make use of the critical-pair analysis technique using *AGG* [11]. Critical-pair analysis provides us with the minimal set of graphs, such that all possible overlapping situations between the left- and the right-hand sides of the rules are recorded. It captures *potential* conflicts and dependencies between rules, rather than (labels representing) transformation steps. Therefore, the result is an over-approximation of the actual dependencies at the level of the labels, specifically where more complex conditions on data values are used. Since we are working at the level of signatures, we represent the overlapping of rules as a relation between the parameters identifying those overlapping graph elements.

Example 3 (conflicts and dependencies on labels). For a small set of labels we have illustrated these relations in Table 1. An entry in a row labelled by l_1 and a column labelled by l_2 represents the relation between l_1 and l_2. Each cell can contain either or both of $\nearrow$ or $<$, be empty or, in case the labels are completely unrelated in either direction, contain $|$.

Referring to Table 1, *addBug*(...) depends on *addProj*(...) since we require a project identified by *project_id* in order to add a bug and therefore $1 = addProj(\text{“}ABC'', \text{“}ERP'') < 101 = addBug(1, \text{“}U'', \text{“}V'')$. Similarly $delProj(1) \nearrow assignProj(1,\ 2)$ as *delProj*(...) would disable the execution of *assignProj*(...). Finally, $11 = addIssue(1, 2, D, E) \mid viewProj(1)$ are independent.

Note that, for future reference, we have included on a gray background relations with some labels to be added in the second version of the service. For the third version, where the *addBug* operation is refined, new dependencies between existing operations are <u>underlined</u> and the output of a label is shown by putting the output value before the body of the label with an equal sign e.g. “$11 = addIssue(1, 2, D, E)''$. In the same way we highlight new parameters added to the existing signatures.

Having lifted information about conflicts and dependencies to labels, we can use this information in two different ways, for filtering out invalid sequences and for defining equivalence classes. If the observation function δ is faithful, we are able to determine if a sequence of labels s is a valid observation.

- If $l_1, l_2 \in s$ such that $l_1 \nearrow l_2$, then l_1 precedes l_2.
- If $l_1, l_2 \in O$ such that $l_1 < l_2$ then l_1 precedes l_2 in every sequence s containing l_2.

Table 1. Asymmetric conflicts and dependencies

First/Sec (↓/→)	1=addProj ("ABC","ERP")	2=addUser ("A","B","t","abc")	assignProj (1, 2)	101=addBug (1,$\underline{2}$,"U","V",$\underline{2}$)	...	delProj (1)	11=addIssue (1,2,"D","E")	updtIssSt (1, 11, 22, "XYZ"	delIssS (11, 22)	delIssue (11)
1=addProj ("ABC","ERP")			$\prec$	$\prec$	...	$\prec$	$\prec$			
2=addUser ("A","B","t","abc")			$\prec$	$\underline{\prec}$	...		$\prec$			
assignProj (1, 2)		$\prec$	$\nearrow$		...					$\prec$
101=addBug[2] (1,$\underline{2}$,"U","V",$\underline{2}$)				$\nearrow \prec$	...	$\prec \nearrow$				
updateBugSt ("ABC","ERP", 1)					...	$\prec$				
delBugSt ("ABC","ERP", 1)					...	$\prec$	$\prec$			
delBug ("ABC","ERP", 1)		$\prec$		$\nearrow$	...	$\prec$				
unassignProj ("ABC","ERP", 1)			$\prec$	$\prec$	...					
unassignBug ("ABC","ERP", 1)					...		$\nearrow$			
delUser		$\prec$	$\nearrow$		...					
viewUser		$\prec$			...		$\vert$			
updtProj (1,"DEF","ERP")					...					
updateUsr ("ABC","ERP", 1)					...					
viewProj (1)	$\prec$				...		$\vert$			
delProj (1)	$\prec$		$\nearrow$	$\nearrow$	...	$\nearrow$	$\nearrow$			
11=addIssue[3] (1,2,"D","E")	$\prec$	$\prec$			...		$\nearrow \prec$			$\prec \nearrow$
updtIssSt (1, 11, 2, "XYZ")					...			$\prec$	$\prec$	
delIssSt (11, 22)					...			$\nearrow \prec$	$\nearrow \prec$	$\prec$
delIssue (11)					...		$\prec \nearrow$			$\prec \nearrow$

In particular, $l_1 \nearrow l_2$ and $l_2 \nearrow l_1$ implies that there is no sequence containing both labels. We denote the set of sequences satisfying these conditions by $O^\circ \subseteq O^*$. Given a finite approximation of dependency and conflict relations, this feature could be used to filter out test cases that are not executable according to the model.

Moreover, we can partition sequences of labels into equivalence classes, called traces, by considering them equivalent if they differ only for the order of independent labels. The set $Traces(\mathcal{G})$ is the quotient of O° under this equivalence. These traces are a generalisation of the classical notion [12] taking into account asymmetric dependencies. Since all sequences in a trace represent the same concurrent behaviour, we can avoid running more than one test from each trace, thus potentially reducing the size of our test suite. However, in this paper we are concerned with the evolution of observable behaviour, not its reduction with respect to a single version of the system.

Example 4 (example of traces through example). With labels as given in Table. 1, a trace $[1 = addProj(``X'', ``Y''); 2 = addUser(``A'', ``B'', ``t'', ``m''); assignProj(1, 2); viewProj(1)]$ contains these additional sequences.

[2] Label updated affecting dependencies.

[3] Label updated without affecting dependencies.

$$1 = addProj(\text{“}X'', \text{“}Y''); 2 = addUser(\text{“}A'', \text{“}B'', \text{“}t'', \text{“}m''); assignProj(1,2); viewProj(1)$$
$$1 = addProj(\text{“}X'', \text{“}Y''); 2 = addUser(\text{“}A'', \text{“}B'', \text{“}t'', \text{“}m''); viewProj(1); assignProj(1,2)$$
$$1 = addProj(\text{“}X'', \text{“}Y''); viewProj(1); 2 = addUser(\text{“}A'', \text{“}B'', \text{“}t'', \text{“}m''); assignProj(1,2)$$

In order to study the effect of evolution of the service on the observable behaviour, we have to consider the changes to dependencies and conflicts on labels. For example, if operations are extended by new features, this will result in more specific pre and post conditions and therefore create new dependencies. But if we introduce new conflicts or dependencies, we will potentially make illegal existing sequences or differentiate between sequences that previously have been equivalent. Where we want to preserve observable behaviour, dependencies and conflicts have to be reflected, i.e., each dependency or conflict in the new version has to be matched by a corresponding one in the old version. This condition for preservation of behaviour at the level of observations has been studied in detail in [9]. In the following section we are going to use it to justify our classification of test cases.

3 Model-Based Evolution

In this section, we first present an evolution scenario in two steps. Then we use the scenario to illustrate our approach to regression test suite reduction.

Evolution Scenarios. In the first evolution step, the Bug Tracker service is extended in order to record Issues with the projects, i.e., concerns raised by users that may not be faults yet indicate deviations from their actual needs. The additional rules and extended type graph are shown in Fig. 4.

In the second evolution step we include a feature to record, among other details, a priority level while adding a bug. That means, the signature of $addBug(\ldots)$ is changed as well as its specification by the rule. Not surprisingly therefore, the modified operation will have additional dependencies, such as $addUser(\ldots) < addBug(\ldots)$. A minor update to $addIssue(\ldots)$ means that the description of the *status* is preset to “First Report”

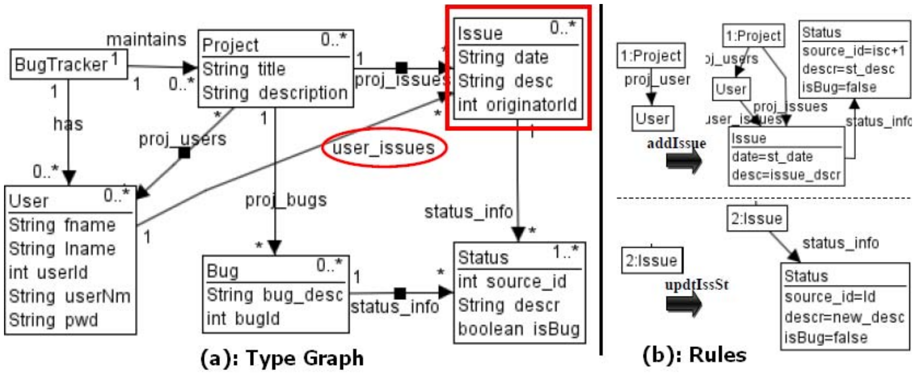


Fig. 4. BugTracker, evolution to Version 2

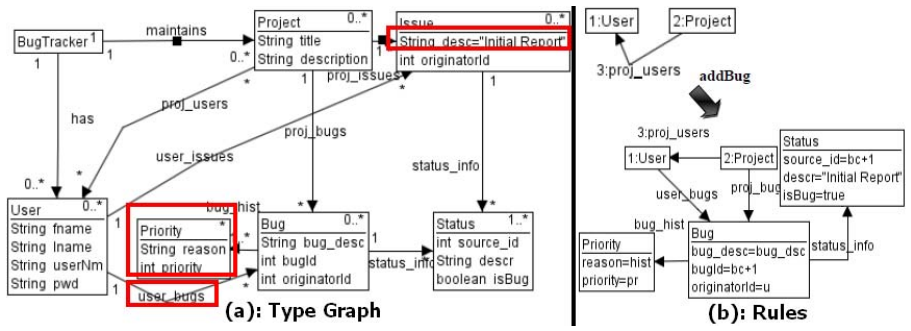


Fig. 5. BugTracker, evolution to Version 3

when the issue is initially reported. There is no change to the signature in this case, and the dependencies and conflicts are not affected. The new version of the changed rule along with the type graph are shown in Fig. 5.

Classification of Test Cases. Given a regression test suite RTS for one version SUT of the system under test, we are going to provide a classification of test cases with respect to an evolution of SUT into SUT' that will distinguish

- *obsolete* test cases OB, that are no longer executable in SUT', either because signatures have changed or because additional preconditions in the model prevent the the execution of operations;
- *reusable* test cases RU, that are still executable in SUT';
- *required* test cases RQ, that are still executable and test new or modified functionality in SUT'.

We will refer to the three versions of our model as $V1$, $V2$ and $V3$.

Traces may become *obsolete* because of changes in the operation signatures. In this case, $O \setminus O'$ are labels that are valid for SUT, but invalid in SUT', e.g., due to missing or incorrectly typed parameters where O' represents the set of labels according to the new version. All traces containing these labels are obsolete as well. In addition, traces could become obsolete because of new dependencies or conflicts emerging in SUT'. The total set of obsolete traces is $O^\circ \setminus O'^\circ$. To see if a sequence s is obsolete in SUT' we have to check (1) if there are any new dependencies $l_1 \prec l_2$ between labels l_2 in s and labels l_1 not preceding l_2 in s and (2) if there are new conflicts $l_1 \nearrow l_2$ between l_2 in s and l_1 occurring in s after l_2. If this is not the case, the sequence remains valid and reusable RU.

In the evolution step $V1 \rightarrow V2$, all conflicts and dependencies are reflected because, while new rules were added, existing rules have not been changed. Hence all traces are preserved and therefore $OB = \emptyset$. For $V2 \rightarrow V3$, both signature and dependencies have evolved. In particular, $addBug(projId, bug\_desc, status\_desc)$ is obsolete, so all traces containing labels based on this operation are obsolete as well. Instead there are new labels based on the extended signature $addBug(projId, userId, priority, bug\_desc, status\_desc)$. We notice that there are new dependencies shown underlined in Table. 1 which render some of the traces obsolete e.g. $addProj(\ldots); addBug(\ldots); viewProj(\ldots)$ was possible in $V2$ but not in $V3$ owing to additional dependency $addUser(\ldots) \prec addBug(\ldots)$.

Test cases in RQ, which exercise operations that may have changed or are affected by changes to other operations, are classified as *required*. Denote by $M \subseteq O \cap O'$ the set of labels such that either their specification or implementation has changed. The set of labels directly and indirectly affected includes M and all labels l_2 such that a label l_1 is affected and $l_1 \prec l_2$ or $l_1 \nearrow l_2$. The set of required test cases is therefore given by the set of all reusable ones RU which contain at least one affected label.

In evolution $V1 \rightarrow V2$, $RQ = \varnothing$ since there are no modifications to existing operations. New test cases will be required to validate the newly added operations, but this is out of the scope of regression testing. Considering $V2 \rightarrow V3$, we find that $\{addIssue(\ldots), \ldots\}$ have been modified and therefore any traces involving their labels are required. Traces containing $addProject(\ldots)$ and $addUser(\ldots)$ are required as well because they have dependency relation with $addIssue(\ldots)$.

4 Evaluation

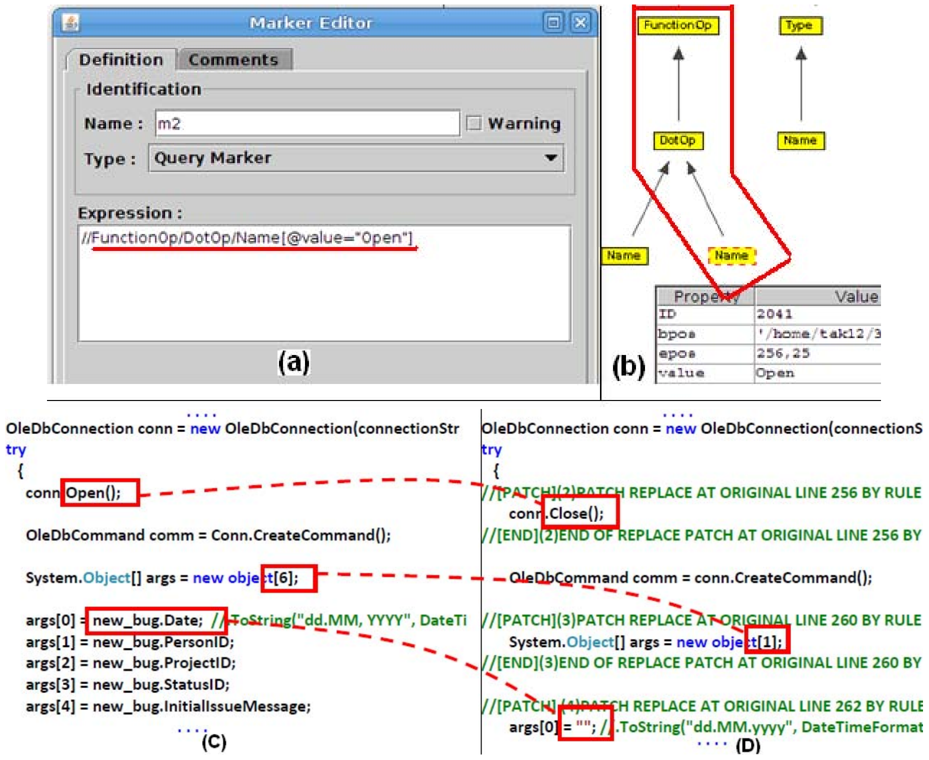
In this section we evaluate, on a small but real example, both the correctness of our method and the reduction in the set of test cases obtained. That means, we will answer the questions: Do the smaller sets of required test cases RQ find the same faults as the larger sets of reusable test cases RU? What is the difference in size between RQ and RU and what would be the smallest test set able to find the faults seeded?

For each evolution step the evaluation is performed in four steps that are outlined below and explained in more detail throughout the section.

1. Generation of test cases.
2. Validation of the quality of the entire test suite.
3. Classification of test cases into OB, RU, and RQ.
4. Validation of the quality and required size of RQ by comparing the results of executing RQ and RU.

We generated test cases manually, based on the information in the model, but without applying a formal notion of coverage. The completeness of the test set is evaluated instead trough *fault seeding*, i.e., deliberate introduction of faults to be detected by the execution of test cases. The percentage of the seeded faults detected provides a statistical measure of the capability of the test set to find similar errors in the system, i.e., a measure of confidence in our test suite [13]. In order to decide which faults to introduce we identified suitable fault types, and then developed rules for seeding them automatically. After applying the rules to the code of the system, we execute the entire test suite to assess its quality. In an iterative process we add test cases until all of the seeded errors were detected.

After applying to the resulting test set the classification described in Section 3, we validate the completeness of RQ against RU by seeding errors into the classes of our service implementation that were modified in the recent evolution step. We then run the tests in both RQ and RU, comparing their results. The evaluation is based on



Name : m2 ☐ Warning
Type : Query Marker
Expression :
//FunctionOp/DotOp/Name[@value="Open"]

Property	Value
ID	2041
bpos	'/home/tak12/3
epos	256,25
value	Open

```
                ....
OleDbConnection conn = new OleDbConnection(connectionStr
try
  {
    conn.Open();

    OleDbCommand comm = Conn.CreateCommand();

    System.Object[] args = new object[6];

    args[0] = new_bug.Date;  //.ToString("dd.MM, YYYY", DateTi
    args[1] = new_bug.PersonID;
    args[2] = new_bug.ProjectID;
    args[3] = new_bug.StatusID;
    args[4] = new_bug.InitialIssueMessage;
                ....
```

```
                ....
OleDbConnection conn = new OleDbConnection(connectionS
try
  {
//[PATCH](2)PATCH REPLACE AT ORIGINAL LINE 256 BY RULE
    conn.Close();
//[END](2)END OF REPLACE PATCH AT ORIGINAL LINE 256 BY

    OleDbCommand comm = conn.CreateCommand();

//[PATCH](3)PATCH REPLACE AT ORIGINAL LINE 260 BY RULE
    System.Object[] args = new object[1];
//[END](3)END OF REPLACE PATCH AT ORIGINAL LINE 260 BY

//[PATCH](4)PATCH REPLACE AT ORIGINAL LINE 262 BY RULE
    args[0] = ""; //.ToString("dd.MM.yyyy", DateTimeFormat
                ....
```

Fig. 6. Fault seeding with L-Care

implementations in C# of the the three versions of the Bug Tracker service. The programming environment Pex[4] has been used for automated unit testing of individual classes. Pex is able to generate test cases based on analysing the source code, with the aim of detecting faults that could lead to runtime errors such as inappropriate exception handling. In our report below we do not include these tests because unit testing is part of the coding at the provider's site while we are concerned with service-level acceptance testing by the client. Therefore, test cases we have generated are concerned with deviations from the public specification of the service interface. We have generated 66 test cases for version $V1$, 83 test cases for version $V2$ and 101 test cases for version $V3$.

Faults are classified by [14], into *domain* and *computation* faults. A domain fault results from control flow errors, where programs follow the wrong path, while a computation fault occurs when the programme delivers incorrect results while following a correct path (usually due to errors in assignments or invocations). More specifically, we have followed the fault types discussed in [15], which also supports calculating a measure of confidence in a test suite. Rules for seeding faults according to these types are implemented in the source code transformation tool L-Care[5], which allows to define markers based on *XPath* queries as shown in Fig. 6(a) on an XML representation

[4] http://research.microsoft.com/en-us/projects/pex/

[5] A product of http://www.atxtechnologies.co.uk/

Table 2. Distribution of seeded faults

Fault Type	# of Seeded Faults			Code Examples	
	V1	V1	V2	Correct Statement	Mutant Statement
Wrong declaration	6	8	9	**new object**[6]	**new object**[0]
Wrong assignment	23	34	35	args[0] = DateTime.Now;	args[0] = " ";
Wrong proc. handling	27	32	35	**throw** ex	//**throw** ex
Control faults	22	27	29	if (conn.Open == ...)	if (conn.Open != ...)
I/O faults	27	32	35	conn.Open()	conn.Close()
Total	105	133	143		

of the code. A sketch of this XML in tree form is shown in Fig. 6(b). Examples of the original and the fault-seeded code are shown in Fig. 6 (c) and (d) respectively. Table. 2 shows the total number of faults seeded for each version as well as a breakdown into the different types along with typical representatives.

We tested all the three versions, extending our test suites until all the seeded faults were detected. Our test cases classification was based on computing a conservative (over-)approximation of the actual dependencies and conflicts between labels using the AGG tool [11]. . Disregarding the data content, we keep track only of the fact that two parameters in two labels are instantiated with the same value. This reduction is safe because it leads to more, rather than less dependencies and conflicts between concrete labels, and thus to more test cases in RQ. In the last step of the evaluation we seed faults in the modified classes of $V2$ and $V3$ only and execute the two sets of required test cases RQ to determine if all of the seeded faults are discovered and how many test cases are actually required to discover them. We have seeded 28 and 18 faults in $V2$ and $V3$, respectively, the smaller numbers owing to the size of the changed classes in comparison to the entire code base. The results are reported in Table. 3.

Table 3. Test case classification and success rate

	$V1 \to V2$		$V2 \to V3$	
Test cases	produced	successful	produced	successful
Obsolete (OB)	0	0	12	0
Reusable (RU)	66	0	45	12
Required (RQ)	0	0	26	12
New (NT)	–	17	–	18

We record the number of test cases in each category *produced* by our classification as well as the number of test cases actually *successful* in finding faults. Of step $V1 \to V2$ we recall that $OB = RQ = \emptyset$ because none of the existing operations were modified. Unsurprising, therefore, none of the remaining test cases in RU found any fault, but 17 new test cases NT had to be produced to detect faults seeded into newly added operations. With the second evolution step, 26 out of 45 existing test cases were classified as required RQ, of which 12 where successful in finding faults. Again, 18 new test cases where added to cover features not addressed by existing test cases. That means, our reduction in the size of test suites has not resulted in missing any faults, i.e., the numbers of faults discovered using *RU* and *RQ* are the same. The reduction in size is significant, but probably still not optimal, because a smaller set of 12 rather than 26 test cases would have been sufficient. This is despite an exhaustive error seeding strategy, which produced faults of the designated types wherever this was possible in the code. The reason could be in over approximation of dependencies and conflicts which, like in many static analysis approaches, leads us to err on the captious side.

To conclude the evaluation, let us discuss a possible threat to the validity of these results. When using the the set of reusable test cases *RU* as a benchmark for the required ones *RQ*, the assessment depends on the quality of the original test suite, which was evaluated by fault seeding. But fault seeding will only deliver relevant results for the types of faults actually sown, while unexpected or unusual faults are not considered. Our approach here was to use approaches to fault classification from the literature, but in order to gather relevant statistics about the costs savings possible we would require data on error distributions from real projects.

5 Related Work

Several techniques [16,17,18] have been using model level information for regression testing. Extended finite state machine (EFSM) are considered in [16], where interaction patterns between functional elements represented by transitions are used for test set reduction. Two tests are considered equivalent if they represent the same interaction pattern. Therefore, whenever a transition is added or deleted, the effect of the model on the transition, the effect of the transition on the model and any side effects are tested for. That means test cases are selected with respect to elementary modifications of the state machine model .

EFSM are also considered in [17] where a set of elementary modifications *EM* is identified. Two types of dependencies, data dependencies *DD* and control dependencies *CD* are discussed. A state dependence graph SDG represents *DD* and *CD* visually and a change in the SDG leads to a regression testing requirement to verify the effect of the modification.

The technique presented in [18] uses UML use case and class diagrams with operations described by pre and post conditions in OCL. A unique sequence diagram is associated with a use case to specify all possible object interactions realising the use case. Changes in the model are identified by comparing their XMI representations.

An approach to regression testing of web services suggested by [2] makes use of unit tests based on JUnit. Test cases are produced by the developer, who generates QoS assertions and XML-encoded test suites and monitors I/O data of previous test logs to see if the behaviour is changed.

[19] constructs a global control flow graph CFG and defines special call nodes for each remote service invocation. A CFG containing a call node, referred to as non-terminal graph, is converted to a terminal graph by inserting the CFG corresponding to that call node. Whenever an operation is modified, the previous and the resulting call graphs are compared to find the differences and all downstream edges are marked as "dangerous" once a modified node is marked.

We make use of semantic information in service interfaces and lift dependencies and conflicts derived to the level of observable actions as they would be seen by a user of the service. Apart from differences in the models used (visual contracts instead of state machines, sequence diagrams or OCL) we employ (asymmetric) dependencies as well as conflicts to characterise admissible sequences of observations. The use of asymmetric relations is due to our richer notion of model, which accounts for data transformation rather than automata-like protocols the order or frequency of method invocations. Dependency information used, e.g., in [16,17] is instead derived from state machines. Pre and post conditions on application data are also used with [18]. While conceptually close, our visual contracts are more easily usable than a textual encoding in OCL and provide a formal operational semantics with a well-established theory of concurrency as a basis for verifying formally the correctness of our approach.

6 Conclusion and Outlook

In this paper we have presented a method to reduce the size of a regression test suite based on an analysis of the dependencies and conflicts between visual contracts specifying the preconditions and effects of operations. The method is applicable to all software systems that have interfaces specified in this way, but is particularly relevant for services because of the lack of access to implementation code and the potential cost involved in running a large number of tests through a remote and potentially payable provider. The method is backed up conceptually and formally by a related paper [9] providing an observational view of the concurrent behaviour of graph transformation systems. In the present paper we have evaluated the approach through the development of a case study showing that (1) the reduced test sets could find all the faults detected by the larger sets while (2) being significantly smaller.

As future work we are aiming to automate the generation of dependencies and conflicts on labels, formalising the over approximation required to represent finitely a relation on an infinite set of labels. We are also working on coverage criteria for test suites based on contract dependencies as well as on a solution to reduce test suites by omitting (the generation of) equivalent test sequences.

References

1. Canfora, G., Penta, M.D.: Testing services and service-centric systems: Challenges and opportunities. IT Professional 8, 10–17 (2006)
2. Penta, M., Bruno, M., Esposito, G., Mazza, V., Canfora, G.: Web services regression testing. Test and Analysis of Web Services, 205–234 (2007)
3. Rothermel, G., Harrold, M.J.: Analyzing regression test selection techniques. IEEE Transactions on Software Engineering 22 (1996)
4. Leung, H., White, L.: Insights into regression testing [software testing]. In: Proc. Conference on Software Maintenance, pp. 60–69 (October 1989)
5. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Monographs in Theoretical Computer Science. An EATCS Series. Springer, Heidelberg (2006)
6. Heckel, R.: Graph transformation in a nutshell. In: Electr. Notes Theor. Comput. Sci., pp. 187–198. Elsevier, Amsterdam (2006)

7. Lohmann, M., Sauer, S., Engels, G.: Executable visual contracts. In: VLHCC 2005: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing, pp. 63–70. IEEE Computer Society, Washington, DC, USA (2005)
8. Lohmann, M., Mariani, L., Heckel, R.: A model-driven approach to discovery, testing and monitoring of web services. Test and Analysis of Web Services, 173–204 (2007)
9. Khan, T., Machado, R., Heckel, R.: On the observable behavior of graph transformation systems. Technical Report CS-10-003, Department of Computer Sciences (August 2010), `http://www.cs.le.ac.uk/people/tak12/observable.pdf`
10. Baldan, P., Corradini, A., Montanari, U.: Contextual petri nets, asymmetric event structures, and processes. Information and Computation 171(1), 1–49 (2001)
11. AGG: AGG - Attributed Graph Grammar System Environment (2007), `http://tfs.cs.tu-berlin.de/agg`
12. Diekert, V., Rozenberg, G.: The Book of Traces. World Scientific Publishing Co., Inc., River Edge (1995)
13. Pfleeger, S.L.: Software Engineering: Theory and Practice. Prentice Hall PTR, Upper Saddle River (2001)
14. Howden, W.: Reliability of the path analysis testing strategy. IEEE Transactions on Software Engineering SE-2(3), 208–215 (1976)
15. Pasquini, A., Agostino, E.D.: Fault seeding for software reliability model validation. Control Engineering Practice 3(7), 993–999 (1995)
16. Korel, B., Tahat, L., Vaysburg, B.: Model based regression test reduction using dependence analysis. In: Proc. Conference on Software Maintenance, pp. 214–223 (2002)
17. Chen, Y., Probert, R.L., Ural, H.: Model-based regression test suite generation using dependence analysis. In: A-MOST 2007: Proc. of the 3rd Intl. Workshop on Advances in Model-based Testing, pp. 54–62. ACM, New York (2007)
18. Briand, L.C., Labiche, Y., He, S.: Automating regression test selection based on UML designs. Inf. Softw. Technol. 51(1), 16–30 (2009)
19. Ruth, M., Oh, S., Loup, A., Horton, B., Gallet, O., Mata, M., Tu, S.: Towards automatic regression test selection for web services. In: COMPSAC 2007: Proceedings of the 31st Annual International Computer Software and Applications Conference, pp. 729–736. IEEE Computer Society, Washington, DC, USA (2007)