Specify Once Test Everywhere: Analyzing Invariants to Augment Service Descriptions for Automated Test Generation

Amit Paradkar* and Avik Sinha

IBM T J Watson Research Center, 19 Skyline Drive, Hawthorne, NY, USA 10532
{paradkar,aviksinha}@us.ibm.com

Abstract. We present a technique which enables a novel *specify once, test everywhere* paradigm by exploiting invariants in a reference ontology. In our approach, each service operation is described in an IOPE paradigm: Input, Output, Precondition and Effect. Our approach augments the service description by creating additional service fault specifications to describe the exceptional behaviors which may arise as a result of invariant violations. We describe our invariant analysis technique and present experimental results which justifies the underlying intuition.

Keywords: Invariants analysis, Service Functional Testing, Automated Test Generation.

1 Introduction

Service oriented architectures (SOA), and Semantic Web Services are specified using standards such as OWL-S and WSDL [1] and consists of service descriptions in terms of Inputs, Outputs, Preconditions, and Effects (IOPEs). Complete specification of semantic service contracts entails specification of exceptional behavior as well. Furthermore, the same fault may be returned by more than one services. Oftentimes, these faults originate from an invariant on the system state. Identification of preconditions for fault messages which originate from system state invariants may be an onerous task for a service specifier. Furthermore, testing of such web services poses challenges because of the need to ensure that services which may violate such invariants have a correct behavior.

In the past, we reported a technique for Automated test generation for semantic web services described using IOPE[7]. However, this technique does not exploit the information provided in the invariants in the reference ontology. In this paper, we introduce a novel approach which exploits the invariants in a reference ontology to augment a service description with additional `fault` messages. In particular, the specific contributions of this paper are:

1. A novel technique to perform analysis of invariants on the reference ontology and the IOPE model of services that manipulate the instances of classes of the reference ontology to derive appropriate additional alternate faults for relevant services.

* Contact author.

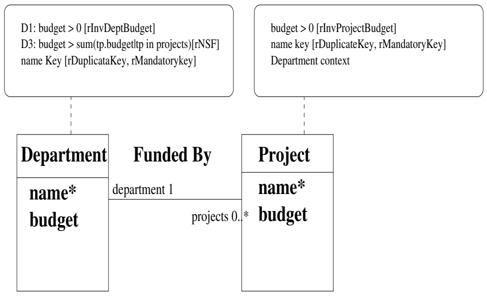
D1: budget > 0 [rInvDeptBudget]
D3: budget > sum(tp.budget|tp in projects)[rNSF]
name Key [rDuplicataKey, rMandatorykey]
budget > 0 [rInvProjectBudget]
name key [rDuplicateKey, rMandatoryKey]
Department context
Department
name*
budget
Funded By
department 1
projects 0..*
Project
name*
budget

Fig. 1. Reference Ontology for `DPSpec`

2. Results of an experiment using several service descriptions which demonstrate the benefits of using such approach during two phases of service development lifecycle: service description and service testing.

The rest of this paper is organized as follows: Section 2 describes our invariant analysis technique. Section 4 reviews the work related to ours and finally Section 5 summarizes the work and provides directions for future work.

2 Analysis of Invariants

Figure 1 shows a graphical representation of the reference ontology for a simple web service, called `DPSpec`, which manages Departments and Projects within a department.

It consists of 2 classes: `Department` and `Project` along with the properties as shown. Classes `Department`, and `Project` have object property `FundedBy` with cardinality constraints as shown. The reference model also has several invariants described using SWRL [3]. For example, invariant labeled `D1` states that the `budget` attribute of the `Department` class has to be positive. Another interesting invariant `D3` states that the `budget` of a `Department` should be sufficient to fund the cumulative `budgets` of its `Projects`.

Figure 2 describes some of the operations in `DPSpec` as specified by the modeler (before the augmented alternate flows derived during invariant analysis). For example, operation `Create Department` takes two inputs: `dName` and `dBudget` and returns as output: `rc`. The pre-condition for the operation is `TRUE` and its effect is of creating a `Department` instance and initializing its attributes to the supplied values. Each operation has a default successful behavior and potentially several faults each representing an exceptional behavior. The operation `Modify Department` has a normal flow guarded by the condition that ensures that a department object which satisfies the search criteria exists; along with appropriate state updates. An alternate flow is executed if no department object satisfying the search criteria is found and results in a `WSDL:fault`. Operation `Move Project` allows to move a project from an existing department to another one.

Information present in the invariants on a reference ontology may not be consistent with the information provided in the service operations model. Assuming that the

```
Operation Create Department
  in String dName
  in Double dBudget
  FLOW rOK IF
    TRUE
    effects {d:=Create(Department)
             d.name := pName
             d.budget := pBudget}


Operation Modify Department
  in String pName
  in String pNewName
  in Double pBudget
  FLOW rOK IF
    ∃ d:Department•d.name = pName
    effects {d.name := pNewName
             d.budget := pBudget}
  FLOW rDeptNotFound IF
    ∀ d:Department• d.name ≠ pName
    fault {message:=No such department}
```

```
Operation Move Project
  in String pD1Name
  in String pD2Name
  in String pPName
  FLOW rOK IF
    ∃ d1,d2:Department,
      p:Project•{d1.name =pD1Name}∧
      {d2.name = pD2Name}∧
      {p ∈ d1.projects ∧
        p.name = pPName}
    effects
      {CreateLink(Fundedby, d2,p)
      DeleteLink(Fundedby, d1,p)}


Operation Modify Project
  ...
```

Fig. 2. Operations for Web Service `DPSpec`

information in the invariants is correct, the inconsistencies could be resolved by either adding more flows (with new guard conditions) to the service operations model or by adding new effect statements to the operation model. In this paper, we focus on the first class of repair actions since these affect the test generation process. The second class of repair action is an indication of a genuine model error that needs to be fixed by the modeler.

2.1 Relating Operations and Invariants

To facilitate the necessary analysis, we need to establish the relationship among the invariants on reference ontology and the service operations since both these artifacts refer to the entities in the same reference ontology. Our invariant analysis technique accomplishes this task by first computing a *tripartite* graph, called `SOROINGraph`, in which the set of operations, the set of Reference Ontology entities(Classes and Properties), and the set of Invariants each form a partition. An operation node has an edge to a reference ontology entity if the operation manipulates (either *creates*, *updates* or *deletes*) the entity. Such an edge is labeled with 1) the nature of the manipulation (Create/Update/Delete), 2) the set of attributes of the concerned entity being initialized/modified, and 3) the *navigation path* - the chain of class accesses - used to reach the instance being manipulated. An invariant has an edge to a reference ontology entity if the invariant refers to it. Such an edge is labeled with the referred attributes and the navigation path used to reach each attribute.

Our analysis exploits the `SOROINGraph` to identify the set of potential operations affected by an invariant. This is done by traversing the edge from an invariant to the reference ontology entities referred in it and then following the edges from the reference ontology entities to the operations that modify those entities. Thus, invariant D1 refers to class Department and attribute budget which in turn are modified by operations Create Department and Modify Department.

2.2 Deriving Flow Conditions

The next step is to create a predicate which reflects the conditions under which the invariant will be violated. We will illustrate the approach through several examples.

Our approach starts with the simplest class of invariants: those which do not involve navigation or aggregate functions. For example, the invariant D1 has the following form: $\forall d1 : \mathit{Department} \bullet \texttt{d1.budget} \geq 0$. Service operation Create Department has a effect statement d.budget = pBudget. Our approach exploits the SOROINGraph to recognize that invariant D1 affects Create Department, and substitutes the variable d for the quantified variable d1 in D1. Furthermore, the assignment of pBudget in the effect statement is accounted for by rewriting d.budget with pBudget. The predicate that leads to violation of the invariant D1 is given by negating the resulting predicate (thus converting the universal quantification into an existential one). The resulting predicate is: $\exists d : \mathit{Department} \bullet \texttt{pBudget} \geq 0$. The expression within the scope of the quantifier is independent of the quantified variable d, and further simplification results in expression $\texttt{pBudget} \geq 0$, which is used as a guard condition for an augmented service operation flow named rInvDeptBudget for Create Department.

The *key* invariants on class C are addressed by identifying the service operations which either create an instance of C (and hence initialize the key attribute) or modify an instance of C by modifying its key attribute. For each such operation, two alternate flows are identified: one which checks for the attribute value being duplicate (for uniqueness) and another which checks for the attribute value being null (for mandatory). Thus, for Create Department, two alternate flows with faults - rKeyExists and rKeyNull - are created with guard conditions: $\exists d : \mathit{Department} \bullet \texttt{d.name = pName}$ and pName = NULL respectively.

The result of applying the rewrite rule for an invariant with aggregate operation SUM (invariant D3 in DPSpec), and which affects a service operation with Create Link effect (Move Project) is given below. The invariant D3 has the following form: $\forall d : \mathit{Department} \bullet \texttt{d.budget} \geq \texttt{sum (tp.budget} \mid \texttt{[tp : Project} \in \texttt{d.projects]})$. The Create Link effect of Move Project is given by Create Link (Funded By, d2, p). We recognize that the association Funded By is accessed in the invariant by the navigation d.projects, thus we substitute all instances of the quantifier variable d in D3 with the instance variable d2 in the effect. Furthermore, the effect of the Create Link with Project p is accounted for by adding the p.budget to the SUM expression.. The resulting predicate is given by: $\exists\, \texttt{d2.budget} < \texttt{sum (tp.budget} \mid \texttt{[tp : Project} \in \texttt{d2.projects])} + \texttt{p.budget}$.

Unfortunately, considering all the predicates (and thus the alternate flows) obtained during this process may lead to infeasible alternate flows. For example, consider the predicate obtained as a result of applying the SUM invariant and Remove Link rule to service operation `Move Project`: `d1.budget` $<$ `sum (tp.budget` $|$ `[tp : Project` $\in$ `d1.projects ])` $-$ `p.budget`. Given that `d1.budget` $\geq$ `sum (tp.budget` $|$ `[tp : Project` $\in$ `d1.projects ])` holds (because the invariant was satisfied before the `Remove Link`), and that `p.budget` $>$ `0` (due to the invariant on the `budget` attribute of `Project` class), the predicate for the computed alternate flow above cannot be satisfied. We use constraint solvers to statically remove such infeasible alternate flows.

3 Experiments

In order to assess the effectiveness and efficiency gained in model description through the invariants analysis we ran an experiment. For the experiment we obtained web service descriptions for five applications with varying net number of operations. The IOPE description for each service, were manually derived. A reference ontology was populated based on the understanding of the systems. Invariants were modeled for each of the services following which "invariant analyses" were performed. The efficiency benefit comes from the reduction in the net size of the descriptions. Specification of the invariants eliminates specification of fault behavior in the individual service descriptions because such information can be automatically populated through invariant analysis. The reduced size of the description increases its maintainability and also brings in a reduction in cost when the service descriptions are manually administered. To measure this effect, physical and logical sizes of the descriptions were measured before and after the invariants analysis. Physical size of a description is measured by counting the total number of tokens in the description whereas its logical size is measured by counting the possible results of invoking the service. This count could be statically determined from a description as follows:

$$\text{logical size} = \sum_{Operations} \text{Number of WSDL faults in an Operation} + 1$$

The percent reduction in size of the description is used to measure the efficiency benefit(T).

$$T = \frac{S1 - S0}{S0} \times 100\%$$

$S0$ is the physical size or the logical size of the description and $S1$ is the size of the enhanced model.

The effectiveness in test generation comes from the fact that the test generation processes are guided by coverage of the model. Thus if, for instance, one is using "boundary value testing" as his test generation process, the coverage criteria is determined by determining the boundaries of the variables of interest. In black box testing of web services, such variables are determined from their descriptions. Thus if the description does not explicitly refer to the variables of interest, they don't become subject of "boundary value testing". The effectiveness benefit(F) of invariant analysis is computed as follows:

$$F = \frac{f1 - f0}{f0} \times 100\%$$

Table 1. Experiment Measurements

Service	No. of Oprns	No. of Invrnts	T_P	T_L	F_{BVT}	F_{ECT}
ATM	3	1	83.33%	28.57%	33.33%	0.00%
Dept Project	4	1	64.71%	33.33%	37.50%	0.00%
Hotel Reservation	4	2	110.34%	50.00%	100.00%	100.00%
Purchase Order	7	2	108.11%	35.71%	28.57%	10.71%
Library	8	2	143.10%	110.00%	31.25%	31.25%

$f0$ is number of faults targeted by a test suite when it is generated using the actual model and likewise, $f1$ is number of faults targeted when the test suite is generated using the enhanced model. In order to evaluate the effectiveness of the invariant analysis on test case generation, we generated test cases for the web service description using boundary value testing(BVT) [4] and equivalence class testing(ECT) [4] using the both the original and the enhanced model. In BVT, the boundary variables are identified by examining the guards on the flows of the operations. The boundary values for such variables were determined based on the types. in ECT, the process is similar, except that the boundary values are determined based on equivalence classes.

The results of our measurements are summarized in Table 1. In Table 1, the suffix BVT denotes the results for boundary value testing and the suffix ECT denotes the results for equivalence class testing. Also, the suffix P denotes the result for physical size and the suffix L denotes the result for logical size.

As is evident from the measurements the efficiency benefits are higher for services with higher number of operations. This follows the intuition that by concentrating the information one can reduce the effort in modeling of individual operations significantly. This strengthens our hypothesis H_T.

Effectiveness benefits, on the other hand, depend on the kind of domain invariants. If the domain invariant does not affect the variables in the guard of the operation but affect its effects, then the F is maximized. This is so because both BVT and ECT generate test cases based on free variables in the flow conditions and therefore they fail to consider the variables in effect of the operation. Following invariant analysis, the flow conditions are modified to include the variables in the effect that are under the influence of some invariants. Consequently, post-enhancement BVT and ECT target higher number of faults.

4 Related Work

Our work of invariant analysis is in the spirit of consistency analysis of specifications. Several works in the area of consistency analysis and fixing inconsistencies have been reported in the past. We briefly review the most relevant ones here. Nentwich *et al.* [5] introduced the concept of consistency analysis for XML documents based on rules defined in a XML based notation called *xlinkit*. The *xlinkit* rules allow users to specify consistency constraints among XML schema elements. *xlinkit* also has a consistency checker component which takes as input an XML document(s) which it checks for violation of the specified consistency constraints. Nentwich *et al.* [6] extended the consistency analysis in *xlinkit* to incorporate *fixing* of the reported inconsistencies. Egyed [2] reported an approach for fixing inconsistencies across a set of artifacts of a UML design model. The artifacts being considered are class diagrams, statechart diagrams, and sequence diagrams.

5 Conclusions and Future Work

This paper presented an approach which enables a novel *specify once, test everywhere* paradigm by exploiting invariants in a reference ontology. In this approach, each service operation is described in an IOPE paradigm: Input, Output, Precondition and Effect. Our technique augments the service description by creating additional service fault The augmented service operation model is then used during subsequent test generation. We described the techniques used in our invariants analysis approach and presented experimental results which justifies the underlying intuition.

There are several future directions we would like to pursue. We would like to conduct empirical studies both in industrial and academic settings to evaluate the our approach along the dimensions of useability and effectiveness. Since our approach is a form of consistency analysis for service models, we would like to extend it to the other class of invariants which need modifications to the *effect* part of the service model.

References

1. T.O.S. Coalition. Owl-s: Semantic markup for web services (2003)
2. Egyed, A.: Fixing inconsistencies in uml design models. In: ICSE 2007: Proceedings of the 29th international conference on Software Engineering, pp. 292–301 (2007)
3. Horrocks, I., Patel-Schneider, P.F., Boley, H., Tabet, S., Grosof, B., Dean, M.: Swrl: A semantic web rule language combining owl and ruleml (2004)
4. Jorgensen, P.: Software Testing: A Craftman's Approach. CRC Press, Inc., Boca Raton (2001)
5. Nentwich, C., Capra, L., Emmerich, W., Finkelstein, A.: Xlinkit: a consistency checking and smart link generation service. ACM Trans. Interet Technol. 2(2), 151–185 (2002)
6. Nentwich, C., Emmerich, W., Finkelstein, A.: Consistency management with repair actions. In: ICSE 2003: Proceedings of the 25th International Conference on Software Engineering, pp. 455–464 (2003)
7. Paradkar, A., Sinha, A., Williams, C., Johnson, R., Outterson, S., Shriver, C., Liang, C.: Automated functional conformance test generation for semantic web services. In: ICWS 2007. IEEE International Conference on Web Services, pp. 110–117 (2007)