Improving Takuan to analyze a meta-search engine WS-BPEL composition

Manuel Palomo-Duarte, Antonio García-Domínguez, Inmaculada Medina-Bulo
University of Cádiz
Department of Computer Languages and Systems, Escuela Superior de Ingeniería
C/ Chile, 1, 11002 Cádiz, Spain
manuel.palomo@uca.es, antonio.garciadominguez@alum.uca.es, inmaculada.medina@uca.es

Abstract

Web Services composition languages, such as the OASIS WS-BPEL 2.0 standard, open a vast new field for programming in the large. But the inclusion of WS-specific instructions presents a challenge for traditional white-box testing techniques. Takuan helps overcome this challenge by implementing an automated workflow which has already generated invariants from the execution logs of simple WS-BPEL compositions, avoiding the usual problems related to modeling impedance that other techniques face. In this work we show how we have extended Takuan with two complementary techniques to analyze more complex compositions with non-scalar variable content despite the limitations of the dynamic invariant generator used, Daikon. We discuss their relative advantages and disadvantages and illustrate them through some of the invariants obtained from a composition which implements a meta-search engine.

Keywords: Web services, service composition, WS-BPEL, white-box testing, dynamic invariant generation.

1. Introduction

Web Services (WS) and Service Oriented Architectures (SOA) are one of the keys to understand Information Technology in the early future. But usually companies need to compose several WS into higher level services in a platform-agnostic way, so languages to program in the large composing WS, like the OASIS WS-BPEL 2.0 standard, are becoming more and more important.

It is unclear though how to accurately test that the orchestrations defined upon this standard satisfy their requirements. We could either use black-box testing, which is only concerned about program inputs and outputs, or white-box testing, which takes into account the internal logic of the program. The latter requires access to the source code, but produces more refined results. But WS-BPEL presents a challenge for traditional white-box testing, due to the inclusion of WS-specific instructions to handle concurrency, compensate scopes, etc.

We propose using information which is as close to the original WS-BPEL process as possible, avoiding most of the problems related to modeling impedance in the intermediate testing models commonly used. Dynamic invariant generation from a set of execution logs has proved to be a successful technique to assist white-box testing of programs written in traditional imperative languages. Let us note that, throughout this work, invariant and likely invariant are understood (as in most related works) in its broadest sense: properties which hold always or in the specified test cases at a certain program point, respectively.

We have shown in our previous work how Takuan can dynamically generate significant invariants relating scalar variables and constants in an adaptation of the well-known WS-BPEL loan example. Takuan takes a WS-BPEL process definition and a test suite specification and outputs a collection of invariants which hold at certain program points in every test case. It also allows for replacing some of the external services invoked with mockups that will behave according to a test suite specification. This way it can execute a composition when some WS are not available or under certain scenarios in which specific predefined responses will be received from the selected WS.

We will detail in this work the improvements made to Takuan to analyze a larger composition which implements a meta-search engine. Takuan previously inherited from its dynamic invariant generator, Daikon, its inability to analyze data structures more complex than an unidimensional scalar vector. We have worked around this issue in two complementary ways, each with its own advantages and disadvantages on the results and performance obtained.

The rest of this paper is organized as follows: first, we introduce dynamic invariant generation and present some of the benefits that it can report to WS-BPEL in particular. Then, after a general description of Takuan’s architecture, we describe the improvements made to Takuan and analyze its results when applied to the Meta Search WS-BPEL com-
position. Finally, some conclusions are offered, along with an outline of our future work.

2. Motivation

The inherent dynamic nature of WS-BPEL implies new challenges for white-box testing. Most traditional white-box testing techniques cannot be directly applied to this language because of its unusual WS-specific instructions, like those for concurrency or scope compensation.

We will present in this section an alternative technique which has seen recently considerable success: dynamic invariant generation. After a general description, we will detail how it can benefit WS-BPEL in particular.

2.1. Dynamic invariant generation

In its broadest sense, as used by most literature related to Daikon [4, 5], an invariant is a property which always holds at a certain program point. Although invariants can be deduced manually, there are systems that automatically generate them. Invariants have a wide array of applications for improving the quality of new and existing programs [4]: debugging, program upgrade support, verification and documentation, among others.

Basically, we can follow a static or a dynamic approach when generating invariants automatically. Static invariant generators are most common: they deduce invariants statically, that is, without running the program. To deduce invariants, the program source code is analyzed, making the system language-dependant. Invariants generated in this way are always correct, but their number and quality is limited due to the inner limitations of the formal machinery which analyzes the code, specially with unusual languages like WS-BPEL.

Conversely, a dynamic invariant generator [4] is a system that reports likely program invariants based on several execution logs. It includes formal machinery to analyze the information in the log files about the variable values at different program points, such as the entry and exit points for functions or loops.

Thus, the dynamic generation of invariants is not based on an exhaustive analysis of the source code of the program, but on a collection of samples of the resulting control and data flows. That is the reason why they are sometimes called likely invariants. False dynamically generated invariants do not necessarily mean bugs in the tested program: they usually rather come from an incomplete test suite. For example, if the program input \( x \) is a signed integer and we only use positive values as test inputs, we will probably obtain the false invariant \( x > 0 \) at some program point. Upon inspection, we would notice it and improve our test suite including cases with \( x < 0 \).

2.2. Benefits to be gained in WS-BPEL

The main advantage of dynamically generating WS-BPEL compositions invariants comes from its inherent generality: a dynamic invariant generator only needs a rough description of the program structure and available variables and some execution logs with the resulting control and data flows.

For that reason, it does not need to model explicitly complex aspects of WS-BPEL such as fault or compensation handling. The use of these features is modeled as just another part of the reported control and data flow.

This contrasts with most white-box approaches until now, which have created detailed simulation models in testing-oriented environments [2], sometimes even translating the WS-BPEL code into a second language to check its internal logic. But, as described above, it is very difficult to simulate accurately all of WS-BPEL’s complex logic. In case any of these features was not properly implemented, compositions would not be accurately tested.

Another important advantage of dynamic invariant generation is that invariants and the test suite from which they are generated can help improve the quality of each other. On one hand, good test suites exercise all of the complex logic of the composition, and thus produce more accurate invariants. On the other, wrong reported invariants can help identifying scenarios not considered in the input test suite.

3. Description of the system

In this section, we describe briefly some of Takuan’s internals. More information can be found in [9]. Takuan is mainly divided into three parts:

Instrumentation step. In this step, we take the original WS-BPEL process definition and add all the required logic so it generates the execution logs with the data and control flow information that we will need later to infer invariants. For that we have created and integrated into our WS-BPEL engine XPath logging extension functions. These functions do not modify in any way the original behavior of the WS-BPEL composition: they just inspect the variable values in that program point. The modified version of the process definition is called the “instrumented composition”.

In this new version of Takuan we have added the capability to control what variables and program points are instrumented and analyzed by adding the proper extension XML attributes to the relevant elements in the original WS-BPEL process definition. We can filter easily the uninteresting variables and obtain more concise results in shorter time this way.
Execution step  This part runs the instrumented composition under the externally provided test suite. Logs generated during its execution are stored and provided to the following step.

Takuan can have the composition invoke actual WS or replace some of the external services with mockups, that is, dummy services which will reply to the composition’s requests with predefined messages. This is suitable for when not all external services are available for testing, or when we simply want to define what-if scenarios under certain predefined external WS behavior.

Analysis step  In this step we use the Daikon dynamic invariant generator to deduce the invariants. It needs a declaration file, which lists all the observed program points and their variables to be considered, and a collection of data trace files, listing the variable values in every execution of an instrumented program point. Takuan has a preprocessor that creates the declaration file and translates trace files to Daikon input format.

4. Proposed techniques

Takuan has already proved able to dynamically generate invariants from the simple loan WS-BPEL composition example [8]. But it is a rather simple example, with only two conditional branches and scalar variables. Daikon can handle these variables directly.

However, most WS-BPEL compositions in the wild are much more complex. So we will consider for the present paper the sample Meta Search process [7], which implements a meta-search engine upon Google and MSN’s offerings and interleaves their results. This is a more suitable example to highlight Takuan’s current strengths and weaknesses, and much closer to real-world WS-BPEL compositions, as it includes non-scalar variables, loops and concurrency.

The main obstacle that we faced while evaluating Takuan under this composition was the fact that most variables had non-scalar content. This is due to the fact that Daikon, Takuan’s dynamic invariant generator, cannot handle directly data structures more complex than an unidimensional vector of scalars.

We have found two complementary ways to solve this problem. Using one or the other will mostly depend on what sort of invariants we are looking for. For the rest of this section, we will use the following example. Let us suppose that in a test case of the Meta Search composition we obtained 3 results from MSN and 1 from Google, resulting in the following contents for the outputVariable variable:

```
<MetaSearchProcessResponse>
  <noResult>4</noResult>
  <noFromGoogle>1</noFromGoogle>
  <noFromMSN>3</noFromMSN>
  <result>
    <url>http://url1google</url>
    <title>Title1google</title>
    <snippet>Snippet1google</snippet>
    <from>Google</from>
  </result>
  <!-- (three results from MSN) -->
</MetaSearchProcessResponse>
```

We will detail further each of these techniques and their advantages and disadvantages below.

4.1 Matrix slicing

This first approach is based on Kvasir, the C++ front-end for Daikon. It has already faced the same limitations imposed by Daikon that we are facing now on Takuan, and solved them by slicing statically-sized $N$-dimensional arrays (where $N > 1$ by default) iteratively into several $(N - 1)$-dimensional variables, until only unidimensional vectors remain.

Looking at the previous example, we would identify 4 two-dimensional variables with each kind of information grouped by result: result[].url[], result[].title[], result[].snippet[] and result[].from[]. Each of these would be split across the 4 results, resulting in 16 variables. This approach has both its advantages and disadvantages:

- It is the most natural approach to certify invariants about particular elements of a multidimensional structure, as they are contained in different variables. For instance, we might want to make sure that whenever Google provides one or more results, the first result returned always comes from Google. We would only need to look at the invariants related to result[1].from[] in the outputVariable variable at the program points which can be only reached if there is at least one result from Google.
- Every variable corresponds to a contiguous set of elements in the original XML tree. This way, we will still be able in the future to filter redundant vector length invariants using the minimum and maximum length restrictions imposed by its XML Schema type information.
- The number of variables grows very quickly, and it only gets worse as we need to handle more dimensions and more elements. We started with 4 two-dimensional variables, and we ended up with 16 unidimensional variables. Only one more level of nesting in the tree
with 2 elements (using slightly varied search parameters, for instance) could result in as many as 32 variables, and this could happen to any variable in any program point.

Daikon’s space and time requirements could increase considerably [4]: more variables mean longer data trace files and, more importantly, many more variable combinations to check. We would have to carefully select the variables and program points to be analyzed to offset these costs.

- As said before, they are separate variables at all effects, and this has implications on the sort of invariants we finally obtain. Daikon does not and cannot check every possible combination of variables at a program point, for complexity and performance reasons. It is limited to producing invariants which relate 1, 2 or 3 variables at most.

This means that Daikon will be unable to generate invariants which span over any multidimensional structure which is sliced into more than 3 variables. Some kinds of invariants, but not all, can be still generated by using new derived variables, like `results.length`, for instance, with the total number of results.

4.2 Matrix flattening

WS-BPEL uses the W3C XPath [10] standard as its default language for describing assignments (both the value and the destination variable). Boolean conditions for loops, and so on. It is very common to use expressions like `a/b` which return the sequence of all `b` elements under any of the `a` children of the current node in document order.

This is precisely the other way we can reduce the dimensionality of a matrix: flattening it into a single unidimensional vector through a predefined traversal order. We’re particularly interested in the original XML document order, so the last indices will vary more quickly: `a[1]/b[2]` comes before `a[2]/b[1]`.

Going back to our running example, the 4 two-dimensional variables would be flattened as 4 unidimensional variables, each with 4 elements. The advantages and disadvantages of this approach contrast clearly with those of matrix slicing:

- Just as the previous approach was naturally suited to generating invariants from particular elements in the original multidimensional structures, this approach can easily produce invariants which apply to all elements in the original multidimensional structure. We would need it to prove that all results came from either Google or MSN, for instance.

- It will still be able to produce invariants for some elements of the unidimensional variables, but only the ones usually selected by Daikon’s heuristics. There is no guarantee we will actually obtain invariants from the part of the vector that we are interested in.

- The original tree structure is lost, and thus the XML Schema type information cannot be applied directly: though minimum length is still directly usable, maximum length would require more processing. Moreover, the final constraint would be all too weak.

- On the other hand, the number of variables is kept constant over the number of elements and levels of nesting. This is quite important, as we have seen how the previous approach can quickly become very costly over more complex cases.

As the most space and time efficient of the two approaches, we could think of using this method by default, and letting the user switch manually to the more expensive method based on matrix slicing later.

4.3 Mixing both methods for high-dimensional structures

The main focus in this paper has been on two-dimensional matrices, which are most common and apply directly to the Meta Search composition, but we could think of a case where neither style would suffice: suppose we were working with a three-dimensional variable `a[],b[],c[]` in which we would want to infer invariants for the set of all `c` elements under `a[1], a[2]` and so on, without considering which `b` they were in.

Slicing would only generate invariants for the `c` elements of a particular `a` and `b` combination, and flattening would collect all the `c` elements into a single vector. So, to obtain the desired results, we would need to combine both: we would slice across the first dimension (dividing the original content into `a[1], a[2],` etc.) and flatten over the second dimension (grouping all `c` inside each `b` into the same vector).

4.4 Comparison of each method through results obtained

We added support both for the basic matrix slicing and matrix flattening methods to Takuan, in order to evaluate their relative performance and the quality of their results. The hybrid variant will be implemented for compositions using three-dimensional variables once the validity of the two basic methods has been established.

Relative performance. Certain Takuan usage scenarios do not require fully detailed invariants for all program
points. We can dramatically speed these up by only considering the parts we are interested in. We measured the running time needed to run 7 test cases and generated invariants from them for each possible combination of analyzing either all variables or only the subset directly related to the composition’s final reply, and/or analyzing either all program points or only the subset 3 levels deep or less into the WS-BPEL process definition’s activity tree.

The test environment consisted of a machine fitted with a dual-core Intel Core Duo T2250 CPU, with 1GiB of DDR2 533MHz RAM and a 80GB 5400rpm HDD. The base system used was a standard GNU/Linux Ubuntu 8.04.1 distribution installation, with its 2.6.24-18-generic default kernel. The active processes during the test suite were mainly those created by the components of Takuan: the Sun 6.0 JRE (with a maximum JVM heap size of 512 MiB), Apache Ant 1.7.0, Perl 5.8.8, Daikon 4.3.4, ActiveBPEL 4.1, BPELUnit 1.0 and GNU time 1.7. During its execution there were no other processes consuming significant CPU time, memory space, or disk throughput. We measured the *wall time* (clock time) required for running the 7 test cases using all the combinations described above. The results are shown in table 1.

Table 1. Running times for each technique and input filtering combination

<table>
<thead>
<tr>
<th>Method</th>
<th>Program points</th>
<th>Variables</th>
<th>Time (mm:ss)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slicing</td>
<td>64 (all)</td>
<td>17404 (all)</td>
<td>7:18</td>
</tr>
<tr>
<td></td>
<td>4720 (selected)</td>
<td>1:19</td>
<td></td>
</tr>
<tr>
<td></td>
<td>12 (3 levels)</td>
<td>2240 (all)</td>
<td>0:43</td>
</tr>
<tr>
<td></td>
<td></td>
<td>704 (selected)</td>
<td>0:22</td>
</tr>
<tr>
<td>Flattening</td>
<td>64 (all)</td>
<td>11412 (all)</td>
<td>3:46</td>
</tr>
<tr>
<td></td>
<td>3888 (selected)</td>
<td>1:01</td>
<td></td>
</tr>
<tr>
<td></td>
<td>12 (3 levels)</td>
<td>1560 (all)</td>
<td>0:28</td>
</tr>
<tr>
<td></td>
<td></td>
<td>624 (selected)</td>
<td>0:19</td>
</tr>
</tbody>
</table>

* Required setting the maximum JVM heap size to 800 MiB to avoid excessive garbage collection. Slightly biased due to minor virtual memory swapping.

As expected, there is a very remarkable difference in the running times required for the matrix slicing and flattening methods while analyzing all the variables in the composition. Once smaller inputs are presented, both methods present similar performance, with flattening still being slightly faster. Nevertheless, in previous work [9] we have already identified other optimizations which could be applied to make Takuan less reliant on manual variable and program point selection. Some of these are filtering away unused variables in each program point, and comparing only variables deduced to be of the same abstract type.

**Invariant quality.** Among all invariants, the subset of the invariants related to the final reply contained in the outputVariable at the exit point of the main sequence shows an interesting contrast. If we use matrix slicing, we obtain these interesting results after some reformatting:

\[
(...).result[1].from[]
one of { [Google], [MSN] }
\]

\[
(...).result[2].from[]
elements one of { "Google", "MSN" }
\]

\[
(...).result[3].from[]
elements one of { "Google", "MSN" }
\]

\[
(...).result[4].from[]
elements == "MSN"
\]

\[
(...).result[5].from[]
elements == "Google"
\]

\[
(...).result[6].from[]
elements == "MSN"
\]

Whereas with matrix flattening, we obtain:

\[
(...).result.from[]
elements one of { "Google", "MSN" }
\]

Both sets of results are similar, but not quite the same. With matrix slicing we have learned that in the only test case which returns 6 results, the last ones from Google and MSN are alternating as we expected. We can also learn immediately that there are no test cases which return no elements, as otherwise the empty list would be in the list of valid values for result[1].from[]): this is a good example of how Takuan helps find incomplete test suites. Though by only considering this single result we cannot conclude anything about the interleaving of Google and MSN as sources for the first 3 results, we could verify it through the invariants in the points where we know there to be at least one result from Google or MSN.

The results from running matrix flattening instead of slicing offer a stronger, and therefore less detailed, invariant: it simply says that the search results either come from Google or MSN, as it generates invariants from all the result sources combined.

Using one approach or the other would then depend on two factors: first, on whether we were looking for invariants on particular elements or on the whole sequence, and second on the time/space constraints imposed: by its very nature, matrix flattening uses up less resources than matrix slicing, as the number of variables required is independent on both the dimensionality and cardinality of the inputs. We could for instance first approach the WS-BPEL composition generally using the faster matrix flattening method, and close in the key variables and program points where we are interested about specific elements using matrix slicing.
5. Conclusions and future work

Web Services pave the way for the future of distributed computing. Their platform independence and high abstraction level help developers orchestrate heterogeneous smaller parts into reliable large systems, and WS-BPEL offers a powerful and cost-effective approach to do so. However, the unconventional and dynamic nature of WS-BPEL is a challenge for traditional white-box testing.

It is very difficult for a simulation model to provide a one-to-one mapping against the full semantics of WS-BPEL. Instead, we propose using dynamic invariant generation. This technique works around the need for a detailed intermediate model by analyzing execution traces generated by a real WS-BPEL engine. We were successful in applying it to the classic loan example in our previous work [9].

We have extended our system, Takuan, to handle a more complex example, which models a meta-search engine. Two techniques to handle multidimensional non-scalar values with Daikon, Takuan’s dynamic invariant generator, were implemented. After analyzing their results, we conclude them to be complementary, as they report different kinds of invariants: matrix slicing works best for individual elements, and matrix flattening is better suited for studying all elements of a multidimensional structure together. However, matrix flattening does offer a definitive performance lead over matrix slicing when manual variable selection is not applied. We will implement a hybrid method later on for compositions with higher-dimensional content in their variables.

Now that Takuan can work with the multidimensional contents of the variables in the Meta Search sample composition successfully, we plan to improve its running time and reduce its memory footprint by implementing previously identified optimizations. Takuan will have Daikon only compare related variables, and will filter out unused variables at each program point. These optimizations would make Takuan faster, reducing the need for manual variable selection. It is also part of our future work to see how invasive the latter optimization would be on the invariants to be obtained, and how to reduce its impact. We also intend to reduce the number of redundant invariants obtained by taking advantage of Daikon’s integration with Simplify, an external simplifier, and using the type information encoded in the process definition’s XML Schema declarations.

Once Takuan obtains the balance between processing requirements and information detail that we desire, we will move on to study the relation between the quality of the invariants generated and the test case suite used. We will use as test subjects several WS-BPEL compositions and their specifications and test with several suites according to different coverage criteria how many of the assertions in the original specifications are inferred by Takuan.

Later on, we will examine the invariants generated by Takuan and verify whether it can improve the results of other WS-BPEL white-box testing approaches.

Acknowledgements

This paper has been funded by the Department of Education and Science (Spain) and FEDER funds under the National Program for Research, Development and Innovation. Project SOAQSIm (TIN2007-67843-C06-04).

References