Automatic Dynamic Generation of Likely Invariants for WS-BPEL Compositions

Manuel Palomo-Duarte[a],[*], Antonio García-Domínguez[a], Inmaculada Medina-Bulo[a]

[a]*Dept. of Computer Science, University of Cádiz, Spain, C/Chile 1, CP 11002*
Email: {manuel.palomo, antonio.garciadominguez, inmaculada.medina}@uca.es
Phone: (+34) 956 01 54 83, (+34) 956 01 57 80

Abstract

The wide adoption of Web Services has led to the development of languages to compose them, like the WS-BPEL standard. In order to check whether the composition works as expected, one common approach is to analyze it and infer functional properties describing its behavior. Traditional approaches for inferring properties in WS-BPEL have been static: compositions are transformed into specialized analysis models based on some formalization. However, this formalization could be inexact due to theoretical limitations or differing interpretations of the standard by implementers. Dynamic invariant generation solves these problems by extracting the properties from actual executions and has been successfully used in popular languages, but not to WS-BPEL yet. In this work, we apply dynamic invariant generation to WS-BPEL, providing innovative solutions for several features that require special consideration, like highly multidimensional values in variables, an advanced type system or unstructured code. We have implemented these solutions in Takuan and evaluated its performance with several compositions of varying complexity. We present the results obtained and a comparative analysis of the efficiency and effectiveness of our solutions. Results show that the solutions are successful in reducing the cost of applying dynamic invariant generation and the number of uninteresting invariants generated.

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

---

[*]Corresponding author.

---

1. Introduction

Web Services (WS) and Service Oriented Architectures (SOA) are one of the keys to understand computing in the next years. As more WS are deployed, languages to program in the large composing them, like the OASIS WS-BPEL (Web Services Business Process Execution Language) standard (**?**), are also becoming more important (**?**). It is therefore necessary to check that their composition behaves as expected (**?**).

Software quality has been an important issue in software development since the so called *Software Crisis*. Many efforts have been made to ensure that software products meet the requirements of the stakeholders, but none has proven definitive (**?**). One approach for checking the behavior of a program is to extract properties (such as block pre-conditions and post-conditions) from its code and compare them against its specifications. In WS-BPEL, most of the approaches in literature are static: they translate the composition into a formalism and use it to derive properties (**?**). However, using a formalism can introduce inaccuracies in the process due to limitations in the model or differences in the way the WS-BPEL standard is interpreted. The WS-BPEL language lacks an unambiguous and formal specification and some important implementation details can vary between vendors (**?**). A dynamic approach could overcome these inaccuracies: while comparing static analysis with testing, Bertolino and Marchetti stated that "the former yield generally valid results, but they may be weak in precision; the latter are efficient and provide more precise results, but only holding for the examined executions" (**?**).

For this reason, we propose using dynamic invariant generation to test WS-BPEL compositions. Dynamic invariant generation generates functional properties from real executions of compositions in a WS-BPEL engine invoking actual WS (**?**). It has proved to be a successful technique for programs written in traditional imperative languages, such as C, C++, or Java. Let us note that, throughout this work, *invariant* and *likely invariant* are understood, as in most related works (**??**), in their broadest sense: properties which a program holds always or under certain test cases, respectively. Dynamic invariant generation can also be seen as a testing technique, as the inferred properties may highlight errors in the program. This matches the

definition of testing by **?** as "the execution of a program with the goal to find errors".

Myers divides testing into two approaches: black-box testing, which is only concerned about program inputs and outputs, and white-box testing, which takes into account the internal logic of the program. This requires access to the source code, but produces more refined results. While composing WS in a WS-BPEL composition, the tester has access to its code and white-box testing is feasible. On the other hand, since the partner services can be changed at any moment and their code is not usually available, these are usually considered as black boxes. Dynamic invariant generation is usually applied in a white-box manner, extracting properties about the internal logic of the program. In our case, these properties will describe the logic of the composition itself.

The main aim of this paper is to answer the research question of whether dynamic invariant generation can be applied to WS-BPEL compositions of the same complexity as those used in the industry. In other words, we are concerned about the applicability of the technique. We address this question from three perspectives: feasibility of the technique, resources needed to run it and the size (manageability) of its output. The feasibility of the technique is shown with the architecture proposed to generate the invariants for WS-BPEL compositions. As for the resources (time and memory) needed to produce the invariants and the number of invariants generated (which may or may not be manageable for a human), they both depend on the settings used to configure our dynamic invariant generation processes (especially the optimizations to handle the particularities in WS-BPEL), so we provide guidelines on how to select them according to our experiments conducted with different compositions.

We have identified the features of the WS-BPEL language that require special consideration when generating invariants, and provided solutions for them. We have provided various mappings to handle the highly dimensional structures in WS-BPEL variables in complementary ways. The advanced XML Schema type system provides more information than traditional type systems, which should be used to suppress redundant invariants. The combinatorial explosion produced by the large number of variables available at certain contexts should be controlled by only comparing related variables and removing empty variables or unused optional content.

While some of these solutions are mere adaptations of formerly implemented techniques, some others imply research contributions in theory: we

3

introduce a new mapping scheme based on XPath principles in subsection 3.3, we restrict the number of invariants produced from variables defined using a *rich* type system (subsection 3.4) and we also applied the dynamically collected information of *variable comparability* to discard variables in parts of the program where they are not used (subsection 3.5.2).

We have implemented these solutions in Takuan (**?**), the only dynamic invariant generator for WS-BPEL. Its automatic workflow takes a WS-BPEL process definition and a test suite specification and outputs a collection of invariants that the program holds. WS-BPEL compositions will usually invoke one or more external partner WS. However, the behavior of these services can vary depending on many factors and may not be under the control of the testers. In order to obtain repeatable results, Takuan allows for replacing the real external WS with *mockups*. In the context of this work, a mockup is a dummy WS that replies to requests with the predefined messages that have been included in the definition of the test case.

Our research method starts by proposing a WS-BPEL specific dynamic invariant generator based on the Daikon system. We show how invariants can be generated for a simple composition and along with two illustrative examples of invariants being useful for testing. Then, we conduct an empirical evaluation of the results obtained when applying Takuan to four different compositions. They are comparable or larger than those found in the available literature, as listed in (**?**). We measure system performance (time and memory) needed to produce the invariants as well as the number of invariants generated. We made experiments using random test suites with different sizes (ranging from very small ones to large ones) and all possible combinations of settings for the processes. The main strength of the research method is that it is based on empirical experiments using a commercial WS-BPEL engine and WS-BPEL code. The different random test suites and the combination of the settings help to improve the validity of the results. On the negative side it is just a quantitative approach with a limited amount of synthetic compositions. Results obtained using a wider range of real-world composition would be desired. Unfortunately, unlike other languages, there is no public repository of large WS-BPEL compositions to draw from (even less compositions annotated with invariants to conduct a qualitative approach). WS-BPEL compositions are normally only for the internal use of the companies creating them, as they usually include sensitive details about the business practices of the organization.

We have performed a comparative study evaluating how our solutions

4

perform when combined in multiple ways. The study shows how the solutions are useful to various degrees: while some considerably reduce the cost of generating invariants, others are specialized in removing redundant invariants.

In our previous conference publications, we presented some of these solutions. This paper unifies and improves their descriptions. In addition, Takuan can now remove variables that have been empty in all the test cases and can preserve more information while flattening highly-dimensional structures. We have also revised Takuan to accept a wider range of WS-BPEL compositions with more advanced message formats: this paper applies it to two new compositions. Using this new version of Takuan, we perform a more extensive and systematic comparative study on the combined effectiveness of the optimizations on larger compositions.

The rest of this paper is organized as follows. Section 2 introduces WS-BPEL and the existing static and dynamic approaches for generating invariants and for testing compositions in this language. Section 3 argues why dynamic invariant generation is suitable for WS-BPEL, discusses the challenges its application presents and proposes solutions for each of them. Section 4 describes Takuan, our framework for dynamic invariant generation for WS-BPEL compositions. Section 5 presents a series of experiments on Takuan to investigate how it scales to larger compositions and measure the impact of the previously proposed solutions in practice. Finally, Section 6 offers some conclusions, along with an outline of our future work.

2. WS-BPEL validation and verification

In this section, we will introduce WS-BPEL and some of the concepts behind dynamic invariant generation. We will then discuss the existing static and dynamic approaches for validation and verification of WS-BPEL compositions.

2.1. Invariant generation

In this section, we will introduce the concepts behind invariant generation and present some of the available static and dynamic approaches for generating these invariants.

An invariant is a property which always holds at a certain *program point*. We consider that every statement s in the program defines two program

points: the program point *before s* denotes the invariants that are true before every execution of *s*, and the program point *after s* holds those true after executing *s*. Note that if *s* is a block statement, there can be other program points nested between the two program points that it defines. WS-BPEL provides two *structured* activities that can be treated as block statements: <sequence> contains activities that are executed one after another, and <flow> contains activities that are run concurrently.

Classical examples are block pre-conditions and post-conditions, that is, invariants which hold right before and after a sequence of statements. There are also loop invariants, which are properties that hold before every iteration and after the last one.

Manually generated invariants have been successfully used to prove the correctness of many popular algorithms to this day. Nonetheless, their generation can be automated, up to a certain degree. These invariants have a wide array of applications for improving the quality of new and existing programs (**?**):

Debugging An unexpected invariant can highlight a bug in the code which otherwise might have been missed altogether. This includes, for instance, the results of function calls with invalid or unexpected parameter values. This way, it can help alleviate the oracle problem (**?**). When the number of test cases increases and no automated oracle is available, manually checking many outputs is a time-consuming and error-prone process (**?**).

Program upgrade support Invariants can help developers while upgrading a program. After checking which invariants should hold in the next version of a program and which should not, they could write it and compare the new invariants with those of the original version. Unexpected differences would indicate that a new bug had been introduced, in a similar way to regression testing.

Documentation Important invariants can be added to the documentation of a program, so any developer will be able to read them while working on it.

Verification We can compare the specification of a program with the actual invariants obtained to check if it has been satisfied.

Unfortunately, manually generating invariants is a long and time-consuming task. For this reason, their generation is usually automated using static or dynamic techniques.

Static invariant generators (**??**) are most common: as their name states, invariants are deduced statically, that is, without running the program. To deduce invariants, the program source code is analyzed (making the generator language-dependent). Invariants generated this way are always correct, but their number and level of detail is too constrained due to the inner limitations of the formal machinery which analyzes the code, specially with unusual languages like WS-BPEL. WS-BPEL has several uncommon features due to its focus on composing WS. For instance, developers may interleave pieces of block-oriented code (using common control structures such as sequences, conditionals and while loops) and flow-oriented code (using links and join conditions to define control dependencies), unlike most languages that only allow for using one style. Other examples include the fact that variables may contain entire XML documents instead of scalar, vector or matrix data, and that compositions may invoke external services whose behavior could change during the invariant generation process.

Conversely, a dynamic invariant generator (**?**) is a system that reports likely program invariants based on several execution logs. It analyzes the information in the log files about the variable values at different locations in the program. So, the generator is not language-dependent: it only needs the log files to be in the proper format.

As shown in Figure 1, the dynamic invariant generation process is usually divided into three steps: instrumentation, execution and analysis. In the first step, the original program is instrumented with new logging instructions that do not modify the values of the variables. When executed, these instructions will produce the information needed for the analysis step, listing the names and values of the variables at the selected program points. Then, in the execution step, the instrumented program is executed under a test suite producing the information needed for the invariant generation. Finally, the resulting *execution logs* are passed to the analysis step to get the invariants.

As a result, dynamic invariant generation is not based on analyzing the source program, but a collection of data flow samples. That is the reason why they are usually called *likely invariants*. False dynamically generated invariants do not necessarily originate from bugs in the tested program, but rather they may 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
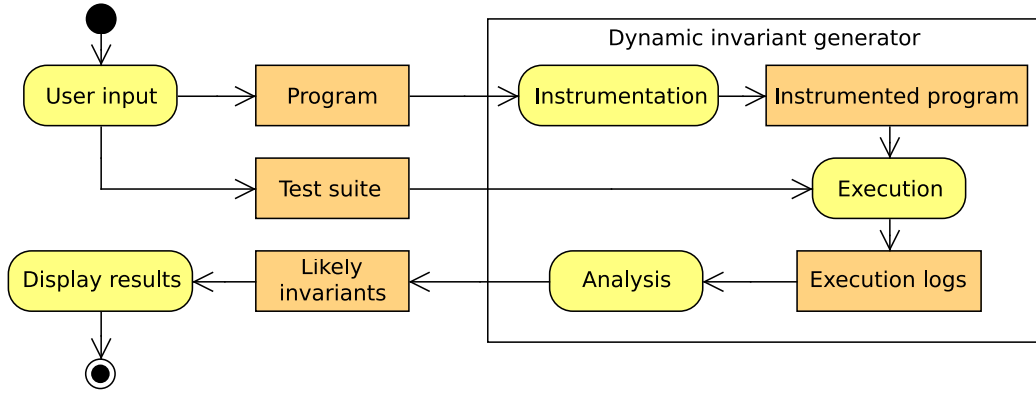
Figure 1: UML activity diagram with the dynamic invariant generation process, adapted from (**?**)

inputs, we will probably obtain the false likely invariant $x > 0$. Upon examination, we would notice it and fix our test suite including cases with $x \leqslant 0$, so the false likely invariant is not inferred anymore.

If we use a *good* test suite (**?**), all of the complex internal logic of the program (loops, function calls, etc.) will be reflected in the collected execution logs, and will help the generator infer true (not just *likely*) invariants. Generally, due to the incremental nature of the process, the more logs we provide the generator with, the better results it will produce (but the longer it will take to get them).

Figure 2 on the following page outlines how to use a dynamic invariant generator: first, the user runs it providing an initial test suite and a program. If unexpected likely invariants are found, the user must check if they are caused by a bug in the code. In that case, the fixed program will be rerun against the same test suite, ensuring that those invariants are not inferred anymore.

But if the user considers the program to be correct, then the user should check if there is a deficiency in the test suite that causes the dynamic invariant generator infer those *false* invariants. In that case the test suite can be refined adding new test cases that produce execution logs disproving them. This way, they would disappear from its output in next run. This feedback loop can be run as many times as needed, fixing bugs in the program code and improving the test suite until the expected likely invariants are obtained.

We can see that this feedback loop defines a new use case for dynamic invariant generation: help improving a test suite. A false dynamically gener-
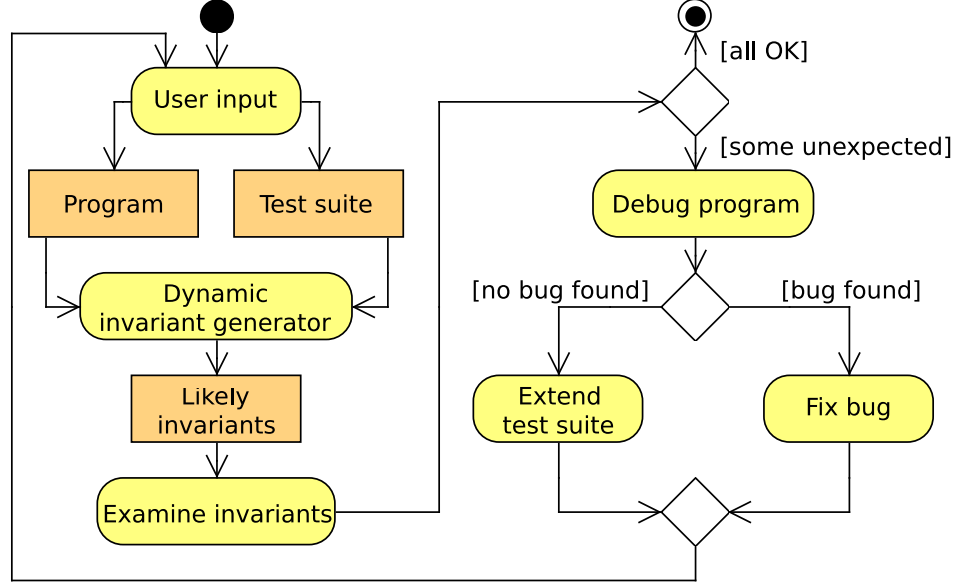
Figure 2: UML activity diagram describing a feedback loop using dynamic invariant generation

ated likely invariant can point out a deficiency in the test suite used to infer it. This way we can add new specific test cases which improve the test suite and expose the false invariant.

Note that from this point on, in this paper we will only refer to *likely invariants*, so we will simply call them *invariants*.

2.2. Related work

In the previous sections, we introduced WS-BPEL and the basic concepts behind dynamic invariant generation. For the rest of this section, we will present other static and dynamic approaches for invariant generation and several alternative techniques for testing and monitoring WS-BPEL compositions.

As stated by other authors, notably **?**, static analysis techniques and testing are complementary techniques. They cannot be compared on the same grounds: "On the other side, static analysis techniques rely on mathematical models of program behavior and infer properties from them, thus complementing testing".

Invariants are recognized in the literature as artifacts belonging to a different realm that test cases. Comparing techniques for creating test oracles

9

and for generating invariants is tantamount to comparing testing and formal verification. Both approaches have their own merits and suffer from well-known limitations. The only connection, in the case of dynamic invariants, is that they are generated from an initial test-suite. In this respect, dynamic invariants are at least as strong as the initial test-suite since they must hold during all the test cases in it; however, they may be invalidated by a new test, unlike the true invariants inferred by static analysis.

Comparing static (true) invariants and dynamic (likely) invariants is difficult, as static invariants are remarkably hard to obtain, particularly in the presence of concurrency, as it is the case with WS-BPEL compositions. To the best of our knowledge, there are no tools that can generate those static invariants for a real concurrent and distributed language such as WS-BPEL.

2.2.1. Invariant generation

There are many approaches for statically extracting properties out of the source code of a WS-BPEL composition. However, most of them are not backed by an automatic tool, according to **?**. Although valid from a formal perspective, they do not scale for actual usage. So we will only focus on two proposals that can be used automatically.

Model-checking is a technique that translates a program into a formal language to check certain properties on it. Using the VIATRA open-source framework, a WS-BPEL composition can be translate into a formal model defined by state transition system, that is later processed using the SAL (Symbolic Analysis Laboratory) tool and verify properties expressed in Linear Temporal Logic (**?**).

WS-BPEL compositions can be easily represented as graphs. This has lead to many works that use Petri nets to check properties. One of the few supported by a tool is (**?**). It uses GNU BPEL2oWFN (part of the Tools4BPEL project[1]) to produce different output files. Using this tool, certain properties of a WS-BPEL execution flow can be checked, such as controllability, generation of the operating guideline, deadlocks or any other temporal logic formula.

As shown above, most static analysis techniques create models of the systems under test (**?**), translating the WS-BPEL code into a formalism. Modeling a WS-BPEL engine and all its underlying infrastructure (operat-

---

[1]`http://www2.informatik.hu-berlin.de/top/tools4bpel/`

ing system, application server, etc.) is very complex, as there is a wide array of non-trivial aspects to be represented. If any of these were not properly translated, compositions would not be accurately analyzed. So we can conclude that this is an error-prone process, as it is not based on the actual execution of the WS-BPEL code in a real WS-BPEL engine while invoking actual services. However, to the best of our knowledge, there are no other dynamic invariant generators for WS-BPEL. For this reason, we will describe the existing approaches for other languages.

Daikon (**?**) is one of the earliest dedicated tools for dynamic invariant generation. It has received considerable attention from the research community, as it is open-source and has support for multiple popular languages through different *front-ends*, such as Java (using Chicory) or C++ (using Kvasir).

InvGen is an automatic linear arithmetic invariant generator for C programs (**?**). Users provide templates of the invariants that they want to generate, and InvGen returns an invariant proving that the error location cannot be reached. InvGen obtains better performance than static linear arithmetic invariant generator thanks to its usage of dynamic analysis.

Purify is a tool that automatically detects memory leaks and access errors in executable programs by instrumenting their execution (**?**). It predates Daikon by a large margin, but it focuses on these two kinds of program properties, rather than on describing the functional behavior of the program.

DIDUCE is a tool that performs dynamic invariant generation and checking during the execution of an instrumented Java program (**?**). Using DIDUCE consists of two steps. First, invariants are generated from a predefined set of "good" executions. Next, the program is run as usual while checking the inferred invariants and reporting any violations. The authors show several cases in which DIDUCE helped debug the root cause for several obscure bugs.

Instead of dynamically checking the generated invariants, as with DIDUCE, another approach is to statically verify them with a third-party tool. Nimmer combined Daikon with ESC/Java in this manner (**?**). Daikon produced a set of likely invariants, which were used to generate ESC/Java annotations in the Java code that could be verified automatically. ESC/Java would then report which of those invariants could not be statically verified.

There is the question of whether the invariants produced by a dynamic approach can be comparable in soundness and exhaustiveness to those produced by a static approach. Nimmer found that Daikon was able to obtain precision and recall values over 90% when compared with manual ESC/-

Java annotations that checked that several Java programs had no runtime errors (**?**).

In summary, we can conclude that most of the existing work is focused on traditional programming languages such as Java or C. In addition, most of the tools are closed-source, except for Daikon and DIDUCE. Finally, there are few tools that maintain a clean separation between the invariant inference process and the logic required to support a specific language, such as Daikon.

2.2.2. Testing

Many testing tools exist for web services. However, most of them only focus on the external service interface and ignore the code that implements the web service. For instance, soapUI (**?**) can generate message fragments from service interfaces and allows for mocking partner services and defining functional test cases as sequences of invocations that depend on each other. All these features are useful for testing WS-BPEL compositions. However, these message sequences and mockups need to be manually specified by the developer, as soapUI does not analyze the WS-BPEL code of the composition.

Testing tools specifically designed for WS-BPEL are much harder to find, as reported by **?**: a more up to date survey is available in (**?**). As an example, the leading WS-BPEL solution from Oracle only implements regular partner service mocking and checking assertions on its outputs (**?**), providing a black-box testing approach. In addition to these, BPELUnit provides parametric testing facilities and generation of coverage reports, being one of the most feature-complete tools to this date (**?**). BPELTester is another tool that combines test path exploration, trace analysis and regression test selection (**?**). These tools allow developers to automate their tests and study their results, but they do not automatically extract likely invariants from the executions of the compositions.

? present a tool that automatically generates test cases for a WS-BPEL composition that meet the state, transition and all-du-path coverage criteria. It is an Eclipse IDE plug-in run as a step-by-step wizard. Similarly, **?** present an add-on for the Oracle tools that generates basis path test suite testing.

As for mutation testing, the only effort up to date is the GAmera mutant generator (**?**). It also provides an evolutionary algorithm for quickly finding strong mutants. An automatic WS-BPEL test case generator based on it is being currently developed. Its operators implement not only the usual mistakes made when programming in traditional imperative languages, but also mistakes that are specific to WS-BPEL.

Finally, the most interesting proposal we have found for regression testing proposes checking the differences between the old and the new versions of the composition and creating specific test cases to test the changed activities (**?**). Nevertheless, it is not backed by an automated tool.

From the above works, we can conclude that only the first steps of an exhaustive WS-BPEL testing process are covered. There are approaches using many techniques, but some of them lack tools to automate them or only support a very limited subset of the WS-BPEL standard. Additionally, beyond automation, the next step should be making the tools interoperable. For example, the test cases obtained from a test suite generation tool should be in a normalized format so they can be accepted as inputs by the available unit testing tools.

3. Dynamic invariant generation for WS-BPEL

In the previous section we introduced the basic concepts behind WS-BPEL and invariant generation and discussed the available approaches for validation and verification of WS-BPEL compositions. In this section, we will present dynamic invariant generation as a suitable technique for generating properties from WS-BPEL compositions, list the challenges presented by WS-BPEL for dynamic invariant generation and propose solutions for them. These solutions will be later evaluated in Section 5.

3.1. Motivation

The inherent dynamic nature of SOA and the uncommon mix of features of WS-BPEL pose new challenges for white-box testing (**?**). Most traditional white-box testing techniques cannot be directly applied to this language because of its uncommon mixture of features, such as concurrency or compensation support, and need to be adapted accordingly (**??**).

As for WS-BPEL, we consider the dynamic generation of invariants to be a suitable technique to support its white-box testing due to:

Use of WS-BPEL code The generator directly uses WS-BPEL code, avoiding errors that could arise in its translation to other testing-specific languages.

Execution in a real WS-BPEL engine All the information in the logs is collected from executions of the WS-BPEL code in an engine. Formally

modeling a WS-BPEL engine and all its underlying infrastructure (operating system, application server, etc.) is very complex, as there is a wide array of non-trivial aspects to be represented. In case any of these was not properly translated, compositions would not be accurately analyzed.

Even more, WS-BPEL is not specified formally, but in (inherently ambiguous) natural language (**??**). For this reason, the formalism used may interpret the WS-BPEL specification differently from a real engine.

Usage of real partner WS Dynamic invariant generation can use the information from the invocations of the actual partner WS. This way, the invariants obtained will reflect the behavior of the composition under real-world circumstances: a partner WS could take too long to answer, could be unavailable, or could even change its implementation (and behavior) while the composition is running.

Nevertheless, this technique can also replace some of the external services with mockups. As mentioned in the introduction, mockups are dummy services which will reply to the composition's requests with predefined messages. Using mockups is only recommended when not all external services are available for testing (due to cost, resource blocking restrictions, etc.), if we want to obtain repeatable results, or when we simply want to define *what-if* scenarios under certain predefined external WS behavior. In any of these cases, the answer provided by each mockup in each test case will be part of the test suite specification given by the user as input. It is up to the user to provide suitable values for them and interpret results consequently.

3.2. Challenges

The first challenge when generating invariants from WS-BPEL compositions is the high dimensionality of most of the variables involved. Most of the existing dynamic invariant generation approaches only deal directly with scalar or one-dimensional data. However, most variables in WS-BPEL compositions contain XML documents, which are arbitrarily complex trees of XML elements. Thus, they cannot be directly mapped, needing preprocessing. We show in Section 3.3 how to deal with this problem by proposing several mapping schemes.

The next challenge is related to the advanced XML Schema type system used in WS-BPEL. This type system may explicitly encode some of the information which would be traditionally only available in invariants generated by the process. For this reason, invariants should only be produced if they are stronger than what is defined by the XML Schema declarations. We further analyze this problem in Section 3.4 and propose a method for suppressing invariants using the XML Schema declarations.

Finally, since WS-BPEL does not implement subroutines, collaboration between activities is done through shared variables. Within this work, a variable is said to be shared between two or more activities if it is available in a context defined by some common ancestor: this may be the entire process itself (making the variable global), or a nested scope (making the variable local). In some cases, all variables may be global, making them available in activities that do not actually use them. This presents two problems. First, a dynamic invariant generator may end up producing many invariants for variables which are not useful in a certain program point. Secondly, the dynamic invariant generator will also take much longer to run, as it will try to extract invariants from each of the many combinations of variables that appear in each program point. We discuss how to solve these problems in Section 3.5.

3.3. Variables with highly dimensional structures

As a WS composition language, variables in WS-BPEL can contain input or output messages from the partner WS. These messages are normally XML documents, which are represented in memory as trees of elements and text nodes. Each element may also contain several attributes. If the tree is sufficiently deep and complex, the elements and their attributes may be nested at many different levels. However, most existing dynamic invariant generation approaches can only generate invariants from scalar values or one-dimensional arrays.

For this reason, it is necessary to map each of these variables containing trees to a set of scalar and one-dimensional array variables. In the process we can preserve some of the original shape of the tree, but some information will be inevitably lost. We will now define and compare two such mapping schemes: *matrix slicing* and *matrix flattening*.

Matrix Slicing maps N-dimensional array variables (where $N > 1$ is a constant in the program) iteratively to several $(N - 1)$-dimensional

15

```
→Sale[1]                      →Sale[1].Order[1].Article      →Sale.Order.Article
    ↪Order[1]                      ↪Article[1]                    ↪Article[1]
        ↪Article[1]           →Sale[1].Order[1].Price             ↪Article[2]
        ↪Price[1]                 ↪Price[1]                       ↪Article[3]
    ↪Order[2]                 →Sale[1].Order[2].Article           ↪Article[4]
        ↪Article[2]               ↪Article[2]                →Sale.Order.Price
        ↪Price[2]                 ↪Article[3]                     ↪Price[1]
        ↪Article[3]           →Sale[1].Order[2].Price             ↪Price[2]
        ↪Price[3]                 ↪Price[2]                       ↪Price[3]
  Sale[2]                         ↪Price[3]                       ↪Price[4]
    ↪Order[3]                 →Sale[2].Order[3].Article
        ↪Article[4]               ↪Article[4]
        ↪Price[4]             →Sale[2].Order[3].Price
                                  ↪Price[4]
```
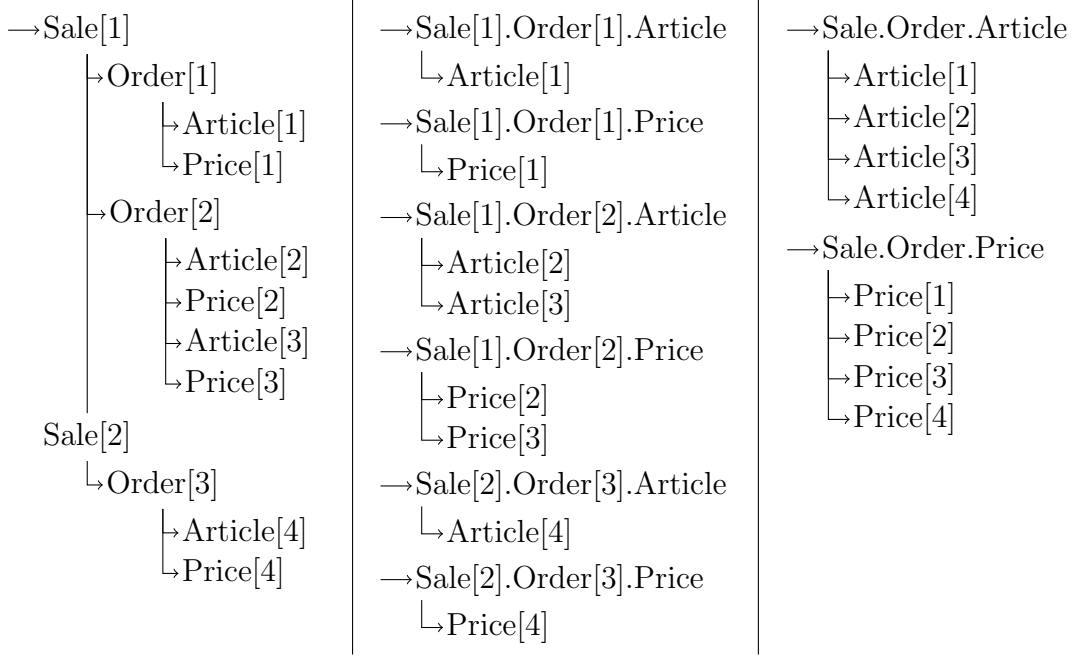
Figure 3: Mapping the input (left) using matrix slicing (middle) or matrix flattening (right)

array variables, until only one-dimensional array variables remain. This approach is based on Kvasir (**?**), the C++ front-end for Daikon.

Figure 3 (left and center columns) shows how it operates on a variable that includes several sales consisting of one or more orders with pairs of articles and their prices. As a result, we obtain two variables for each order in each sale: one for the articles, and another for the prices.

Matrix Flattening reduces the dimensionality of an XML tree by mapping it to a single one-dimensional array for every kind of leaf element. Each of these arrays contains all the children elements of that kind in document order. It keeps the original XML document predefined traversal order, so the last indexes will vary more quickly: in our example *Order[1]/Article[2]* comes before *Order[2]/Article[1]*. Figure 3 shows how this mapping is applied to the running example. We obtain only two variables: one for all the articles in every sale, and another for their prices.

16

As the most space and time efficient of the two approaches, Takuan uses this method by default. Users can switch manually to the more expensive method based on matrix slicing if desired.

This method is based on the W3C XPath (**?**) standard for querying XML documents. WS-BPEL uses XPath as its default language for describing assignments (both the value and the destination variable), Boolean conditions for loops, and so on. In fact, matrix flattening is the result of applying an expression of the form of *//fieldX* for every field of the variable. This expression returns all the elements of type *fieldX* in document order. Using other XPath queries could provide different mapping results: studying this possibility is part of our future work.

3.4. Advanced type system

A dynamic invariant generator normally requires a specification file declaring the variables to be studied, among other inputs. Normally, the specification assigns types to these variables. The existing approaches usually implement limited type systems: for instance, all integers have the same minimum and maximum values. This is suitable for traditional structured programming languages. However, the XML Schema type declarations used by WS-BPEL can be much more specific. For instance, existing types can be specialized into new types with additional restrictions: minimum and maximum values or lengths, enumerations of valid values, and so on.

XML Schema declarations are additional sources of information for suppressing redundant invariants. On the other hand, an XML Schema declaration can have many optional elements that may not be used in practice. These unused optional elements may not be interesting for the user, and will still increase the time required by the dynamic invariant generator. We will dedicate the rest of this section to these two issues.

3.4.1. Suppressing invariants with type system information

The additional information provided by a richer type system should be used to suppress invariants as early as possible in the invariant generation process, in order to obtain the maximum performance gains. For this reason, it should be used in the dynamic invariant generator itself.

In addition to their type, variables need to be annotated with additional restrictions, such as valid ranges of values or lengths, or an exhaustive list of valid values. For instance, if XML Schema tells us that the value of a

17

certain variable must be greater than 100, an invariant confirming it would only add noise to the resulting list of invariants. These restrictions can be extracted from the variable definitions in different programming languages: XML Schema declarations for WS-BPEL, statically sized arrays and matrices in C/C++, fixed length strings in FORTRAN, or `VARCHAR(N)` fields in SQL, for instance. Even more, if that information relates to a multidimensional variable that is later mapped into n unidimensional variables, we will have avoided generating not one, but n redundant invariants.

3.4.2. Dealing with unused optional elements

WS-BPEL variables can have many optional elements and attributes. However, these optional elements and attributes can make even small inputs unmanageable, since they may contain many elements, each with their own optional subtrees and attributes. The dynamic invariant generator will spend a considerable amount of time checking invariants that relate the empty subtrees. So, it may be useful in some cases to know whether these are being actually used in a certain composition or not.

To solve this, before invoking the dynamic invariant generator we can remove all the subtrees that stay empty at a program point throughout all tests from the execution traces and the variable specifications. This is not a regular optimization, as we are not simply removing redundant invariants. We are intentionally discarding potentially useful invariants that we are not interested in. Users should be warned about this when using this optimization.

3.5. Collaboration through shared variables

WS-BPEL does not support the concept of a subroutine. Activities in a WS-BPEL composition collaborate with each other by using variables defined in a common context (i.e. "sharing" them, as mentioned in Section 3.2). In fact, other activities may also be able to access those variables even if they do not use them. This will increase the number of uninteresting invariants that the dynamic invariant generator can produce.

WS-BPEL variables can contain large amounts of information. It may not make sense to check for invariants every pair of subtrees of the same type. For instance, relating article quantities with durations would normally not make much sense, even if they are both stores in fields defined as integers. In addition, an activity will typically only use specific subtrees of certain variables: checking invariants for only those subtrees could further reduce

the cost of the process. In the rest of this subsection, we will describe how to implement these optimizations.

3.5.1. Detecting related variables

We are interested in finding the variables (or parts of them) that have the same abstract type. For example, an integer could represent a monetary sum or the age of a person, among other things. We are only interested in checking invariants that relate monetary sums among them, and ages independently. Variables that have the same abstract type are usually inferred by seeing which of them are used together in the program: an expression like $a + b$ usually means that a and b are of the same abstract type.

There are static and dynamic approaches to finding these relationships. Static approaches, such as Lackwit (**?**) for C, may not be powerful enough for WS-BPEL, as its conditional and assignment expressions can traverse the trees stored in the variables in complex ways. A dynamic approach, like the one in DynComp (**?**) for Java, is more suitable for WS-BPEL. DynComp represents abstract types using *comparability indexes*. The dynamic invariant generator will only try to relate variables with the same index.

To compute these indexes, it is necessary to register which variables are used together in the expressions within the executions of the WS-BPEL composition. First, we need to divide expressions into nested *comparability scopes*: only the paths visited in the same scope will be considered to be related. For example, the following expression checks if the ages of a client that asks for a loan and her guarantor are below a certain threshold, and if a credit card is solvent.

$$(\max(\text{client}[i].\text{age}, \text{guarantor.age}) < \text{threshold}) \wedge \text{solvent}(\text{passport}, \text{visaCard})$$

When processed, it will be parsed into the abstract syntax tree in Figure 4, while creating new comparability scopes at:

- each argument of every function call (such as *solvent*), except for some functions known to take arguments of the same abstract type (e.g. *max*),

- logical operators such as *and* ($\wedge$ in the figure), *or* or *not*, and

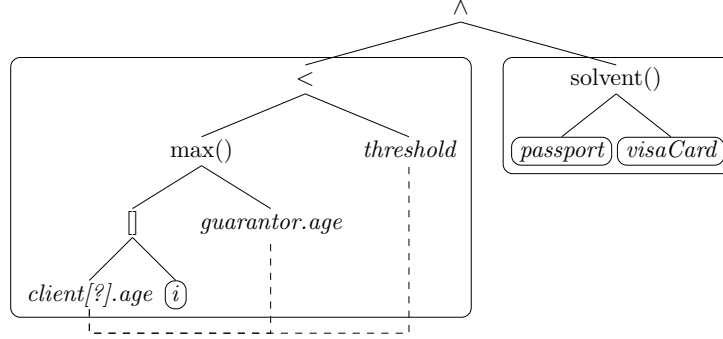- filtering predicates, such as the scope for the variable i in the figure.

Figure 4: Abstract syntax tree of an XPath expression annotated with comparability scopes (rounded rectangles). Variables are in italics. The three related variables are linked with a dashed line, remaining the rest of them in separate comparability scopes.

The execution environment can keep track of which parts of each variable are used within each comparability scope. After running the composition with the above expression, we would find that only three of its six variables, *client.age*, *guarantor.age* and *threshold* were evaluated within the same scope and are actually related. Therefore, there would be four different abstract types in this expression: one for ages, one for the i index of the clients list and two for the *passport* and *visaCard* arguments of *solvent*.

This way, we can calculate related variables in each expression in the program. But variables are related transitively along all the expressions in the program flow. In order to aggregate all the information from the expressions that have been evaluated throughout the composition, we will create a graph using the variables in the specification as nodes. Every time two variables are evaluated within the same comparability scope, we add a link between them. After processing all the execution traces, the abstract types in the composition can be retrieved by listing the connected components of the graph.

3.5.2. Removing unused variables

Normally, each activity in a WS-BPEL composition only uses some of the subtrees of certain variables. It may be useful to only generate invariants for these subtrees and ignore the rest.

If a variable is used in an activity, the dynamic invariant generator may also consider that it is used in its ancestors. This will increase the number of variables to be considered in the ancestors, but will also produce invariants
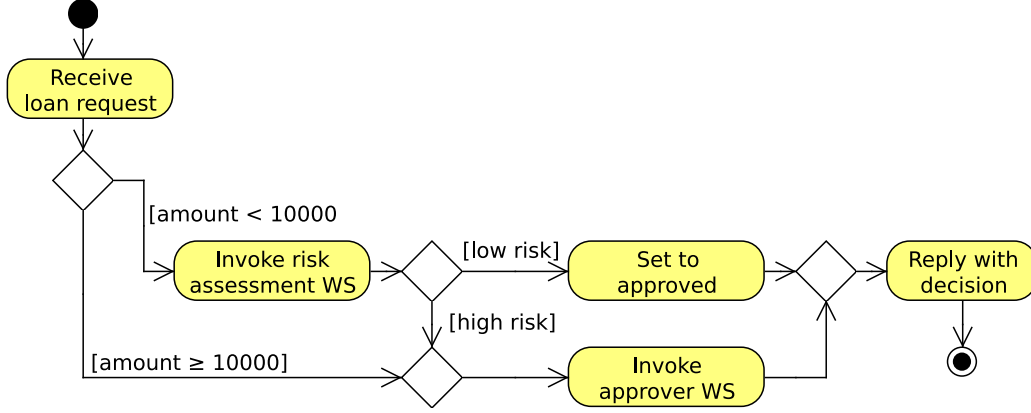
Figure 5: Loan Approval composition represented as a UML activity diagram, adapted from the ActiveVOS InfoCenter.

that consider the combined effects of several more specific activities.

To implement this optimization, we can reuse the results produced when detecting related variables. These results list the variables that were used in each expression in the WS-BPEL composition. In addition to these logs, we will also need to register the variables used to send and receive messages and invoke services within each program point, among other activities. In fact, this optimization can be applied to other languages that allow using shared variables.

4. WS-BPEL dynamic invariant generation framework

This section introduces Takuan, the dynamic invariant generator for WS-BPEL developed in this work. We describe its architecture and each step of the process in the next subsections. We illustrate the process using the version of the loan approval WS-BPEL composition from (**?**) shown in Figure 5 as a UML activity diagram, since WS-BPEL does not have a standard graphical notation.

The composition receives loan requests from customers. Each request includes an amount and certain personal information. The WS-BPEL composition simply notifies the costumer whether the loan request has been approved or rejected. The approval of the loan is based on the requested amount and the risk assessed by an external WS from the personal information of the customer. If the amount is below $10,000 and the risk is low the loan is directly approved.

21

In case the amount is below the threshold but risk is high, the composition invokes an external loan approval WS, and its answer is passed on to the customer. Finally, if the requested amount is over the threshold no risk checking is done, and the composition simply forwards to the client the reply of the external loan approval service.

4.1. Takuan architecture

Takuan integrates our own code with several well-tested open-source systems that have been modified to create a WS-BPEL dynamic invariant generation workflow (see figure 6 on the following page for a simplified diagram): the ActiveBPEL WS-BPEL standards compliant open-source engine (**?**), the BPELUnit unit test library (**?**) that includes built-in support for WS replacement with mockups when desired and the Daikon dynamic invariant generator (**?**).

Takuan basically takes a WS-BPEL process definition and a test suite specification and automatically outputs a collection of invariants which hold at certain program points in every test case. This process is an adaptation of the generic one described in section 2.2.1, that consists of three main steps: instrumentation, execution and analysis.

We must take into account that Daikon was originally designed with traditional structured and object-oriented languages in mind: normally, invariants are generated before and after class methods, functions or procedures, but there are no such concepts in WS-BPEL. Instead, Takuan selects by default all <sequence> and <flow> activities: these represent sequences of activities and sets of concurrent execution branches, respectively. This way, Takuan checks for invariants even inside conditionals, loops, and around external service invocations. By default, all sequence and flow activities and all variables are studied, but users can manually limit them.

4.2. Instrumentation step

In order to extract invariants from actual executions, we need to collect logs with information about which activities are executed and the values of the variables before and after every execution of each activity of interest.

For that reason, we have to take the original WS-BPEL process definition and build a new version that adds this information to the logs. We call the resulting WS-BPEL process the *instrumented version* of the process, and it is functionally equivalent to the original composition. The instrumented version invokes a new set of XPath logging functions that we have extended
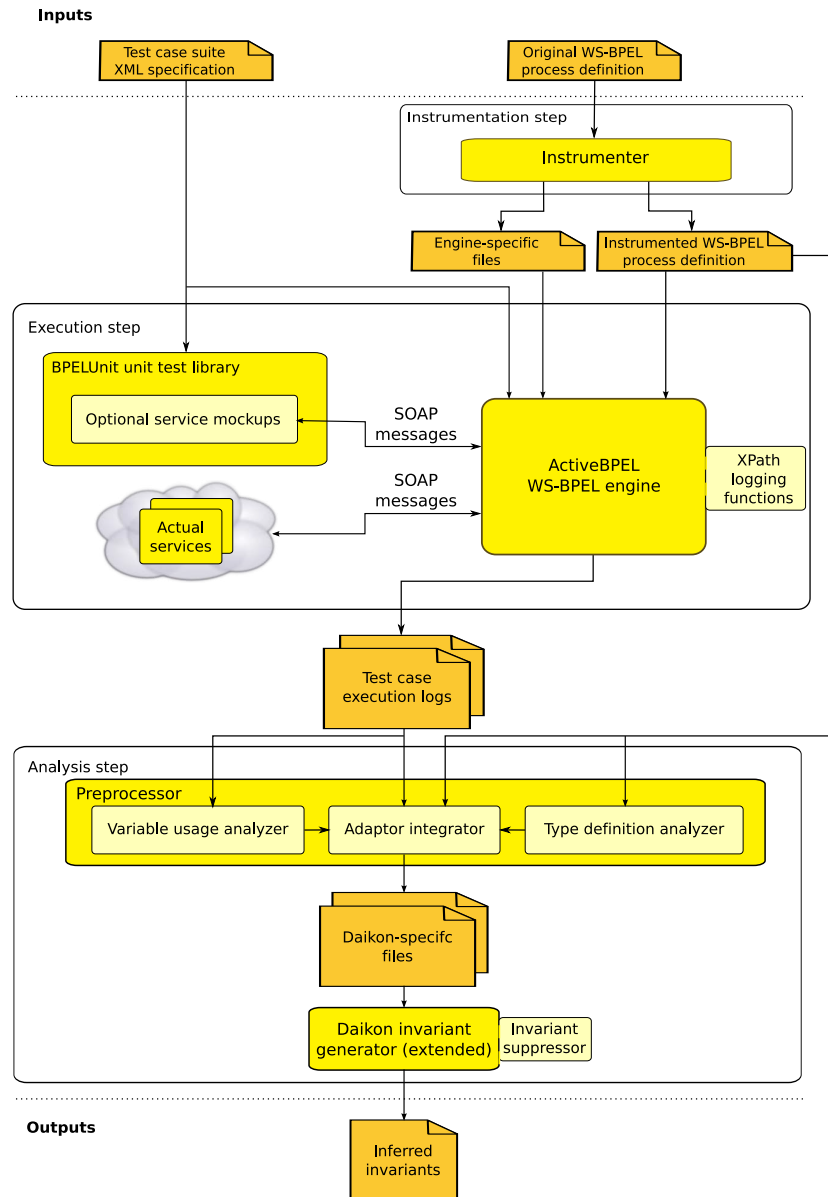
Figure 6: Simplified architecture of Takuan

Listing 1: Simplified excerpt of an ActiveBPEL execution log

INSPECTION($processOutput.accept) = false

Executing [(...)/ sequence/assign]
Completed normally [(...)/sequence/assign]

INSPECTION($processOutput.accept) = true

ActiveBPEL with. These functions only log the values of the variables, while preserving the control flow of the original process.

4.3. Execution step

We have obtained the instrumented WS-BPEL process definition with the required logging logic and the engine-specific files in the previous step. We will now run the instrumented process against the externally provided test suite. Logs generated during its execution will be handed over to the next step.

First, we need to deploy the instrumented WS-BPEL process definition. If the user has requested it, the WS will be replaced with mockups. For instance, in a certain test case for a loan approval composition, a mockup of the risk assessment WS may reply saying that the risk is high in 50% of the cases, and low in the other 50%, in order to test the behavior of the composition in that situation.

Then the test suite will be run, invoking the composition and generating an execution log for each test case therein. Finally, the process will be undeployed. BPELUnit takes care of all these tasks, except for executing the WS-BPEL composition itself, which is up to ActiveBPEL.

A simplified excerpt of an execution log is shown in listing 1. First, the original value for `accept` is inspected, reporting a *false* value. The second assignment in the WS-BPEL sequence, part of the original process definition, sets the variable `processOutput.accept` to *true*, indicating that the loan has been accepted. And finally, the last inspection confirms the change in the inspected variable.

4.4. Analysis step

In the previous step, each test case in the test suite generated its own execution log. The final analysis step will pass the logs through our custom

Listing 2: Some simplified invariants generated by Takuan

```
1  approverInput.amount = processInput.amount
2  approverOutput.accept = processOutput.accept
3  approverInput.amount = 150000
4  approverOutput.accept one of { 0, 1 }
```

preprocessor and hand the results to our own modified version of Daikon, so that it generates the desired invariants.

Our strategies for removing unused optional elements (Section 3.4.2), detecting related variables (Section 3.5.1) and discarding unused variables (Section 3.5.2) are coded in the *Variable usage analyzer*. It applies them to the test case execution logs processing before invoking Daikon, saving CPU time this way. Additionally, the *Type definition analyzer* creates a file with the invariants enforced by type system information that do not have to be checked (Section 3.4.1). Unfortunately this information cannot be incorporated in Daikon input files. So we had to code a *Invariant suppressor* module and plug it to Daikon.

We can see in listing 2 some of the invariants produced by Takuan using an intentionally limited test suite. These invariants follow the textual output format defined by Daikon. They were obtained at the end of the branch that handles large loans (over $10,000$). At that program point, it successfully deduced (see line 1) that the originally requested amount was correctly sent to the approver external service. It also demonstrates (line 4, where true and false values are represented by 0 and 1, respectively) that the approver external service does not approve every loan, and that its answer is used for the final output of the composition (line 2).

However, the results of the generator could be further improved. For example, it tells us in line 3 that the amount requested is always $150,000$, which we know to be false. To understand why this false invariant was produced, we must consider that the invariants for a program point are inferred only from the subset of the test cases which reach it. In our composition, only the test cases with amounts over $10,000$ reach this point. And, in the test suite we used, every test case in this subset asked exactly for $150,000$, so Daikon inferred that false invariant. In general, we can ignore a false invariant if we can prove, as we just did, that it originates from an incomplete test suite and not from a defective WS-BPEL process definition. A more

25

Listing 3: Simplified fragment of a WS-BPEL process definition containing a bug

```
1  <if name="IfLowAmount">
2    <condition>( string(assessorOutput.risk) != 'high' )</condition>
3    ...
4    <else>
5      ...
6  </if>
```

Listing 4: Simplified invariants that highlight the bug

```
1  LoanApproval.SmallAmountHighRisk:::ENTER
2  assessorOutput.risk = "low"
3
4  LoanApproval.SmallAmountLowRisk:::ENTER
5  assessorOutput.risk = "high"
```

robust approach is to add more test cases asking for different amounts over $10,000 so the false invariant is not generated.

This is an example of how Takuan can help identify deficiencies in a test case as well as provide informative invariants. Next, we briefly comment an example of how the invariants obtained in Takuan can highlight a bug in the code. Let us suppose that, by mistake, we change the condition that decides whether the approver has to be invoked or not in a small loan. The relational operator changed from *equal* to *non-equal*, implementing a *if the risk is not high* condition instead of a *if the risk is high* (as shown in listing 3).

In listing 4 we can see the invariants that highlight the bug when entering the two possible branches of that conditional activity. The first one is for small loans with high risk (i.e., the *LoanApproval.SmallAmountHighRisk* program point), and we can observe that in every execution the risk is set to be low. And the other invariant shows the opposite: the sequence that handles low risk loans (the *LoanApproval.SmallAmountLowRisk* program point) is being executed only when the risk is high.

4.5. Getting Takuan

Takuan is freely available under the terms of the GPL 2.0 license. A self-contained command-line distribution with all required components is avail-

able from its website at (**?**). There are other variants, such as an Ant task and a graphical NetBeans plug-in, but the command-line distribution is the most powerful and easiest to install and use.

The main command-line utility reads a YAML-based (**?**) configuration file with all the required information for running the tests and performing the analysis. YAML is a data serialization language that is easier to read and write by humans than XML. Configuration files can range from simple cases that only run a single analysis, to complex configurations that repeat the execution and analysis steps with several combinations of flags. Takuan is smart enough to reuse intermediate results and save disk space as much as possible, by analyzing the dependencies between all the requested executions and analysis.

5. Evaluation

We have so far shown that Takuan can successfully generate meaningful invariants from WS-BPEL compositions. This leaves us with the question of how Takuan would scale to larger test suites and more advanced compositions, with many more program points and more complex content in its variables.

For this reason, we have conducted a series of studies using the setup described in the next subsection. We have compared the time required by each of the steps of Takuan as the number of test cases increases or as the mapping scheme and instrumentation options change. We then perform a comparative analysis between all the possible combinations of optimizations, summarize our findings and present a set of guidelines on how to select them.

5.1. Experiment setup

To study the impact of a larger test suite, we ran Takuan on the following compositions[2]:

- *LoanApproval* (LA), the previously commented loan approval composition. The composition is 68 XML tags long.

---

[2]Technical details and additional figures available at `http://neptuno.uca.es/files/takuan2-perf-exp/`.

- *LoanApprovalExtended* (LAE), a considerably extended version of the previous composition taking into account much more information. It is $1,478$ XML tags long.

- *MetaSearch* (MS), a composition that implements a meta-search engine, aggregating results from the Google and MSN search engines. It is 508 XML tags long.

- *SquaresSum* (SS), a composition computing $\sum_{i=1}^{n} i^2$ for some n. It is 47 XML tags long.

These compositions are comparable or larger than those found in the available literature, as listed in (**?**). Unfortunately, unlike other languages, there is no public repository of large WS-BPEL compositions to draw from.

Since we need to emulate a wide range of *what-if* scenarios for the partner services, we have replaced the partner services in these compositions with mockups. This has allowed us to automatically generate a large number of test cases in which the mockups behave in different ways, by assigning uniformly distributed random values to the inputs of the composition and the variables that controlled the behavior of the mockups. These helped ensure that the different parts of the composition would be exercised, making the invariants more representative. For LA, the risk assessment and approver partner WS reply back with predefined risks and decisions. For LAE, the mockups for the Social Security WS, debtor status WS and other personal information WS provide predefined results about the requested person. For MS, the mockups of the Google and MSN search engines return predefined sets of results that are expected to be combined in certain ways. SS is the only composition that does not invoke external WS, and therefore does not need any mockups.

5.2. Performance by step

We measured the *wall times* (the sum of CPU time and I/O waits) taken by each step in the process implemented by Takuan. The test suites ranged from 5 to $1,000$ randomly generated test cases. Table 1 collects the times for the worst case with $1,000$ tests.

The instrumentation times did not depend on the number of tests, and took less than 5 seconds on all compositions. We can conclude that it scales well to larger compositions without posing any important CPU time or memory size problems.

Composition	I	EO	EI	A
LA	1.19	300.98	633.76	15.31
LAE	4.90	1,652.70	4,867.52	505.39
MS	1.60	320.77	1,259.75	838.37
SS	0.58	310.23	644.09	314.25

Table 1: Time required by each processing step in Takuan for the worst case and $1,000$ tests (in seconds). I is instrumentation, EO is execution without instrumentation (the original version), EI is execution with instrumentation and A is the analysis step (preprocessor and Daikon).

While inspecting every variable in every activity, running the instrumented versions of LA and SS (the simplest compositions) took about twice as long. Instrumentation has a larger impact in LAE and MS (increasing the execution times by a factor of 3 and 4, respectively), but it is still within the same order of magnitude. In practice, this impact will probably be largely reduced, as users will only be interested in some specific variables at certain activities.

Finally, we can see that the relation between the time required by the execution and analysis steps varies from composition to composition. The analysis step requires much less time than the execution step for LA and LAE, and takes up a reasonable amount of time for SS. For MS, the cost of the analysis step is much higher than running the original version of the composition. This indicates that the data used in MS is harder to analyze with Daikon than those used in LA, LAE and SS.

5.3. Performance by mapping

Previously, we studied the impact of each processing step in the total running time of Takuan. We will now study the impact of using a specific mapping scheme and selecting only some activities for instrumentation.

We measured the time needed to run Takuan against each composition, using randomly generated test suites with enough tests to produce a stable set of invariants. Results are shown in Table 2. LA required 50 test cases, LAE required $5,000$, MS required $8,000$ and SS required 500 test cases. We ran every test twice, disabling all optimizations Takuan implements. On the first run, we instrumented all activities. On the second run, we instrumented only the top 3 levels of the activity tree of the WS-BPEL process definition.

29

Composition	Tests	All levels		Top 3	
		Flat	Slice	Flat	Slice
LA	50	48.47	44.86	47.48	44.84
LAE	5,000	26,664.20	17,755.88	26,669.78	17,755.84
MS	8,000	10,513.90	5,208.89	15,484.32	8,225.88
SS	500	399.49	401.22	397.12	398.32

Table 2: Execution times for Takuan in seconds, by mapping scheme (F for matrix flattening and S for matrix slicing) and number of levels analyzed in the activity tree

Each composition shows slightly different results:

- LA and LAE take slightly less time when limiting the number of instrumented activities, but changing the mapping scheme does not have a noticeable effect. The structure of the composition is complex enough to merit limiting the number of activities to analyze, but the complexity of the data used does not require the more efficient matrix slicing mapping scheme.

- SS is too simple for these optimizations: neither the mapping scheme nor selecting only some activities to analyze seems to make a difference.

- MS is the only composition for which both optimizations have a noticeable effect. The total wall time is reduced from $15,484$s with matrix slicing and all activities to only $5,208$s with matrix flattening and 3 levels of activities.

From the above results, we can conclude that selecting certain activities or a specific mapping scheme for performance reasons is only worth it if the composition has sufficiently complex control flow or data structures.

5.4. Performance by optimization

In this section we will study the impact that each of the optimizations proposed in Section 3 have in the analysis of the four compositions described in Section 5.1. We will use each of them separately and in combination, noting that removing unused variables depends on detecting related variables.

We measured the number of variables checked by Daikon, how many invariants were produced and time and memory consumption. Due to space

constraints, we have kept the individual values as supplementary material for the present paper. The box plots in Figure 7 summarize the impact of these optimizations on each of the measured metrics.

As a general comment we can highlight that in case none of our optimizations were applied, Takuan results would hardly be useful for a human. For example, in the MS composition an execution without optimizations provided almost $100,000$ invariants across all program points when using the matrix slicing mapping scheme. The same mapping with all our optimizations activated only produced $1,270$ invariants for all program points.

In the following subsections, we will study the impact of each optimization across all compositions as it is activated, and then analyze the impact of their combinations.

5.4.1. Usage of XML Schema constraints

Memory requirements are normally reduced (by up to nearly 50%) or do not change (as shown by the first three quartiles of its box plot in Figure 7). However, there is the case of the MetaSearch composition when using matrix slicing and evaluating all activities, in which memory requirements were increased nearly by 150%. This is probably due to the high number of invariants which had to be suppressed by Daikon using the XML Schema information. Time requirements do not change much, ranging from a 20% reduction to a 20% increase.

What is striking, however, is the considerable reduction in the number of invariants produced. Looking at the box plots, we can see that the reductions range from slightly over 20% to more than 80% of the original invariants. With matrix slicing, by only applying this technique, we have been able to suppress in some cases more than $9,000$ invariants. Most of these were array lengths, which were already encoded in the XML Schema declarations. In contrast, this technique happens to be much less effective when using matrix flattening, as it requires weakening or removing most XML Schema maximum array length constraints.

5.4.2. Removing empty variables

The box plots in Figure 7 indicate that this optimization usually removes 20.4% of the variables. However, this does not change the memory used by Daikon or the number of invariants very much: there is not much to say about variables which are always empty in every test. This optimization does reduce execution times, however: processing time is considerably reduced
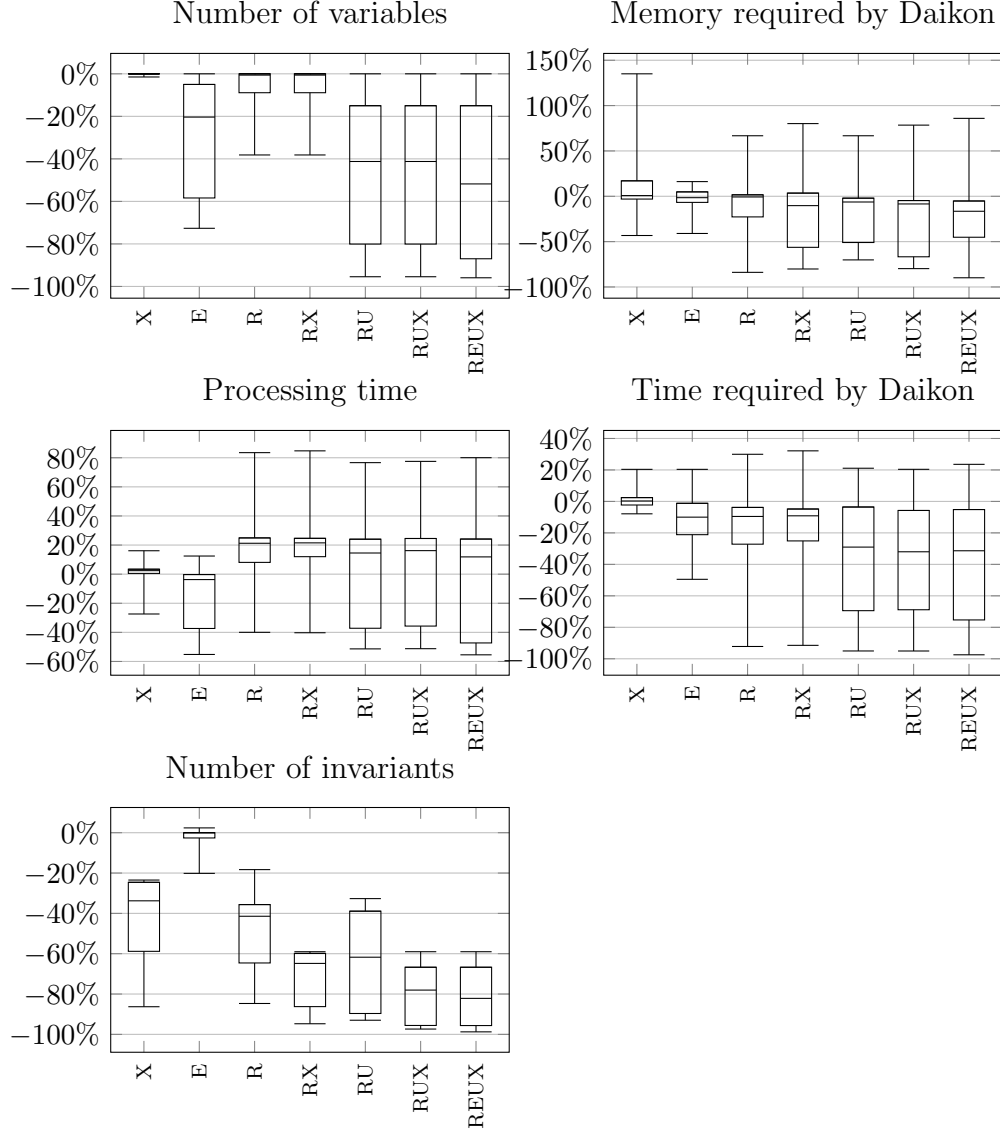
Figure 7: Aggregated changes (as positive percentages of increases or negative percentages of reductions over the original amount) made by each combination of Takuan optimizations on number of variables, space and time performance and number of invariants generated on the four compositions used in the experiments. In this graph, R is "detect related variables", E is "remove empty elements", U is "remove unused variables" and X is "suppress invariants known from XML Schema declarations".

most of the time (up to 55.2%) and so does the time required by Daikon (up to 49.5%).

5.4.3. Detecting related variables and removing unused variables

We must take into account that when this feature is enabled, Takuan ignores activities which do not use any variables. This is done under the assumption that these activities do not introduce any interesting behavior on their own. This explains the reduction in the number of variables in Figure 7. It is normally negligible, but in LAE with 3 levels of activities variables are reduced by up to 38.2%.

From the box plots, we can see that its impact on performance highly varies depending on the composition and the mapping scheme used. The optimization is much more effective in the largest compositions, LAE and MS, with reductions on memory usage of 83.9% (for LAE with matrix slicing and 3 levels of activities), 68.5% and 67.5% and only 2 increases of 20.9% (for LAE with matrix slicing and all activities) and 1.5%. The rest of the time, it usually imposes up to a 20% overhead in memory usage.

Most of the time, this optimization increases the time required by the preprocessing stage of the analysis step in about 20%. Daikon usually takes slightly less time after activating this optimization, but it can have a large impact in some cases: MS with matrix slicing went from 1 hour and 7 minutes to only 9 minutes.

The number of invariants is always noticeably decreased: from 18.3% to 84.7%, with a median of 41.4%. The change is more noticeable on the most complex compositions.

Finally, let us compare R with RU (R removing unused variables). With RU, variables are usually reduced by 41.2% instead of the negligible change with R. Reductions in memory and time are much more common (as shown by the enlarged boxes), even though there are still cases in which they increase noticeably. Finally, the number of invariants is drastically reduced: the median is now at 61.7% instead of 41.4%.

5.5. Combining optimizations

Finally, let us compare the combinations of the above optimizations. Comparing the R and RX (R with XML Schema constraint information) entries from Figure 7, they have about the same performance, but RX generates far fewer invariants. It appears that RX should be always used instead of R.

Likewise, *RU* and *RUX* (*RU* with XML Schema constraint information) have about the same cost, but *RUX* tends to produce fewer invariants. The difference is smaller in this case.

Finally, *RUX* and *REUX* (*RUX* with empty variable removal) also share the same levels of performance, but *REUX* can generate slightly fewer invariants. This is only noticeable in MS, in which the minimum reduction goes from 86.8% to 95.5% and the maximum reduction goes from 97.4% to 98.8%.

Clearly, this is a case of diminishing returns as more and more optimizations are combined. The largest jump in performance seems to come from detecting related variables (*R*), and then from removing unused variables (*RU*): after those, additional optimizations will only reduce the number of invariants produced, without any large changes in the obtained performance.

5.6. Lessons learned and practical implications

From the results of the above experiments, we can extract a set of guidelines on how to best prepare the composition and configure Takuan:

- According to Table 1, running the tests against the WS-BPEL composition can take up longer time than the actual analysis if we instrument every variable in every activity. If execution time is an issue, users should tell Takuan the subset of variables and activities that should be considered to generate invariants.

- If the variables contain multiple levels of nested elements, the user will need to decide whether to generate invariants for specific positions (using matrix slicing) or for the entire sequence at the same time (using matrix flattening). Table 2 shows that matrix slicing can be more expensive: when in doubt, matrix flattening should be used.

- Finally, the user needs to select the optimizations to be used to improve performance and reduce the number of redundant invariants. These optimizations were presented in Section 3.2 and analyzed in the first subsections of Section 5.4.

  By default, Takuan detects related variables, removes unused variables and uses the XML Schema declarations to suppress invariants. These three optimizations can be considered to be safe. However, removing empty variables is not enabled by default, as it may discard potentially

34

useful invariants. Therefore, it should only be enabled if this sort of invariant is not desired, or if the composition requires it due to performance concerns. This can happen when highly complex XML Schema types with many optional elements and many levels of nesting are used.

In short, users should focus on picking the most suitable mapping scheme first and run Takuan with the default options. If the process takes too long or too many invariants are produced, users should then select the most interesting activities and/or variables. If we do not want invariants about empty variables or if it still takes too long, users will need to tell Takuan to remove empty variables, in addition to the default set of optimizations.

This way, the feedback loop proposed in Figure 2 on page 9 (subsection 2.1) can be implemented for WS-BPEL compositions. WS-BPEL compositions are "usually developed using agile and iterative implementation methodologies to quickly adjust to the business process improvements and changes" according to **?**. Most agile methodologies include early testing phases, usually concurrently with development. In this situation, a WS-BPEL programmer could start testing a simple initial version of the composition by asking Takuan to generate invariants concerning all variables in every activity. Using a large random test suite and the above "safe" optimizations would be recommended. This resulting invariants will provide a general overview of the composition. Then, in each development iteration, a regression test could be performed by simply choosing the activities that have been modified. If unexpected likely invariants are found, the user would have to check if they were caused by a bug in the code and fix it. Then, running Takuan once more against the same test suite should provide the expected invariants. In the case the developer believed the composition to be correct, additional and more specific test cases should be added to disprove the unexpected invariants.

In latter modifications of the composition, only the specific activities and variable fields that were affected could be selected to minimize the impact of a more detailed mapping scheme (matrix slicing). For larger changes that affected many activities, more aggressive optimizations could be used at first to quickly check the overall behavior of the composition, and then intensive inspections could be performed on specific activities that were suspect of having bugs.

Finally, in case the tester thought that the external WS changed their implementation, the composition could be run once more against the same

test suite, only inferring invariants in the program points located right after the interaction with the WS. A detailed look into invariants obtained would confirm or deny it.

6. Conclusions and future work

WS-BPEL language allows for programming Web Services in the large, introducing additional logic on top of the existing ones. As a result, WS-BPEL developers have to carefully check not only the partner services, but also the internal logic of the composition. To do this, one possible approach is to analyze the composition and extract properties about its behavior. Among the available static and dynamic approaches, we have selected dynamic likely invariant generation as it is based on actual executions of the WS-BPEL code in a real engine.

In this work, we have answered the research question of whether dynamic invariant generation can be applied to WS-BPEL compositions of the same complexity as those used in industry from three perspectives: the feasibility of the technique, the resources needed to run the process and the size (manageability) of its output.

The feasibility of the technique has been shown in the architecture of Takuan, the only dynamic invariant generator for WS-BPEL. A sample composition was used to show simple invariant usage for detecting bugs in a composition and improving a test suite. The need to handle tree-structured variables has resulted in a new contribution to the theory in dynamic invariant generation. The simple mapping scheme for handling matrices was insufficient for variables with tree structure: a new mapping scheme based on XPath was introduced that could preserve the original order of the elements in the tree and relate them together.

Another three contributions to the theory relate to improving resource usage and output size. We have reduced the number of invariants produced from variables defined using information in their XML Schema definitions. In fact, the theoretical foundations of our proposal can be also applied to other programming languages with detailed restrictions on data types, such as FORTRAN or SQL. Additionally, we have proposed using the information concerning variable comparability that was dynamically collected during executions. This is not just for discarding invariants relating nonsensical combinations of variables, but also for discarding shared variables in activities where they were not actually used.

We have conducted an empirical evaluation of the results obtained when applying Takuan to four different compositions. By examining the collected data, we can confirm that the resources (time and memory) needed to produce the invariants and the number of invariants generated highly depend on the settings used to configure our dynamic invariant generation processes (especially the optimizations to handle the particularities in WS-BPEL). In fact, in some cases an execution without optimizations can provide thousands of invariants, that would hardly be useful for a human. Several guidelines have been provided on how to select optimizations.

Results show that the optimizations produce a noticeable effect in the most complex compositions, but sometimes they hardly have any impact in the smallest ones. In general, all the optimizations reduce the number of invariants produced, though they do not always improve performance. The use of XML Schema restrictions generally reduce memory requirements by over 50%, being highly dependent on the mapping scheme. For matrix slicing, it suppresses in some cases around $9,000$ invariants, while matrix flattening produces fewer invariants which were harder to suppress. As for removing empty variables, it reduces significantly execution costs but does not noticeably change the number of invariants generated. Detecting related variables become more effective with larger compositions, reducing invariants by between 18.3% and 84.7% and decreasing running times, in exchange for more memory. As for removing unused variables, it reduces the number of invariants drastically: nevertheless, it must be noted that there are disadvantages to this approach, because some invariants concerning the global behavior of the composition might be discarded.

Finally, we propose using Takuan in a feedback loop so that WS-BPEL compositions developed using agile methodologies can meet Q&A criteria. Starting with a simple initial version of the composition and a large random test suite the developer can use the invariants to implement regression testing in consequent development iterations. Then, depending on the modifications in each iteration different settings are recommended for invariant generation.

Regarding future work, we have three lines of work ahead. Firstly, the matrix flattening mapping scheme could be generalized. As commented in subsection 3.3, matrix flattening is based on applying a certain generic XPath query to a multidimensional matrix to obtain n unidimensional variables. Takuan could be extended to accept an specific XPath-like query for each multidimensional variable. This way, testers could use customized XPath expressions to obtain invariants concerning specific fields of their interest for

each WS-BPEL variable.

Secondly, we will analyze the relation between the test suite and the quality of the generated invariants. We could use as test subjects several WS-BPEL compositions and manually calculate formal specifications. This way, we could use several test suites meeting different coverage criteria to see how many of the assertions in the original specifications are inferred by Takuan. This information could be used to create a knowledge base for an expert system that provides advice to improve a test suite.

Finally, we could study the applicability of genetic programming and mutation testing to help generate better invariants, as proposed by **?**. The technique produced promising results in Java, but it has not been applied to WS-BPEL. In fact, it proposes an algorithm that ranks invariants according to their ability to tell apart the original program from a changed version (a mutant). This approach could be used as the base of an expert recommendation system.

All the information needed to replicate our work (compositions, test suites, configuration files and Takuan, version 2.0) is publicly available in the following address[3].

Acknowledgments

References

Bartolini, C., Bertolino, A., Elbaum, S., Marchetti, E., 2009. Whitening SOA testing, in: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, ACM, New York, NY, USA. pp. 161–170.

Ben-Kiki, O., Evans, C., döt Net, I., 2009. YAML Ain't Markup Language (YAML) Version 1.2. `http://www.yaml.org/spec/1.2/spec.html`. Last checked: January 4th, 2014.

---

[3]`http://neptuno.uca.es/files/takuan2-perf-exp`

Bertolino, A., Marchetti, E., 2005. A brief essay on software testing, in: Thayer, R.H., Christensen, M. (Eds.), Software Engineering, The Development Process. Wiley-IEEE Computer Society Press, third edition.

Bjørner, N., Browne, A., Manna, Z., 1997. Automatic generation of invariants and intermediate assertions. Theoretical Computer Science 173, 49–87.

Bozkurt, M., Harman, M., Hassoun, Y., 2013. Testing & verification in service-oriented architecture: A survey. Software Testing, Verification and Reliability (STVR) 23, 261–313.

Braione, P., Denaro, G., Pezzè, M., 2012. On the integration of software testing and formal analysis, in: Bertrand Meyer, M.N. (Ed.), Empirical Software Engineering and Verification, Springer. pp. 158–193.

Bucchiarone, A., Melgratti, H., Severoni, F., 2007. Testing service composition, in: Proceedings of the 8th Argentine Symposium on Software Engineering (ASSE'07).

Castro-Cabrera, C., Medina-Bulo, I., 2011. An approach to metamorphic testing for WS-BPEL compositions, in: Proceedings of the 2011 International Conference on e-Business, SciTePress, Seville, Spain. pp. 137–142.

Colón, M., Sankaranarayanan, S., Sipma, H., 2003. Linear invariant generation using non-linear constraint solving, in: Jr., W.A.H., Somenzi, F. (Eds.), Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings, Springer. pp. 420–432.

Davis, M.D., Weyuker, E.J., 1981. Pseudo-oracles for non-testable programs, in: ACM '81: Proceedings of the ACM '81 conference, ACM, New York, NY, USA. pp. 254–257.

Domínguez-Jiménez, J., Estero-Botaro, A., García-Domínguez, A., Medina-Bulo, I., 2011. Evolutionary mutation testing. Information and Software Technology 53, 1108–1123.

Dumas, M., Hofstede, A.H.M.T., Russell, N., Verbeek, H.M.W., Wohed, P., 2005. Life after BPEL, in: 2nd International Workshop on Web Services and Formal Methods (WS-FM 2005), volume 3670 of Lecture Notes in Computer Science, Springer-Verlag. pp. 35–50.

Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D., 1999. Dynamically discovering likely program invariants to support program evolution, in: Proceedings of the 21st international conference on Software engineering, ACM, New York, NY, USA. pp. 213–224.

Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D., 2001. Dynamically discovering likely program invariants to support program evolution. IEEE Transactions on Software Engineering 27, 99–123.

Ernst, M.D., Czeisler, A., Griswold, W.G., Notkin, D., 2000. Quickly detecting relevant program invariants, in: ICSE 2000, Proceedings of the 22nd International Conference on Software Engineering, Limerick, Ireland. pp. 449–458.

Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C., 2007. The Daikon system for dynamic detection of likely invariants. Science of Computer Programming 69, 35–45.

EviWare, 2012. SoapUI. `http://www.soapui.org`.

Gupta, A., Rybalchenko, A., 2009. Invgen: An efficient invariant generator, in: Proceedings of the 21st International Conference on Computer Aided Verification, Springer-Verlag, Berlin, Heidelberg. pp. 634–640.

Hallwyl, T., 2008. Evaluating the BPEL standard specification. `ftp://ftp.diku.dk/diku/semantics/papers/D-609.pdf`. Master thesis.

Hangal, S., Lam, M.S., 2002. Tracking down software bugs using automatic anomaly detection, in: Proceedings of the 24th International Conference on Software Engineering, ACM, New York, NY, USA. pp. 291–301.

Hastings, R., Joyce, B., 1991. Purify: Fast detection of memory leaks and access errors, in: In Proceedings of the Winter 1992 USENIX Conference, pp. 125–138.

Heffner, R., Fulton, L., 2007. Topic overview: Service-oriented architecture. Forrester Research, Inc.

Kovács, M., Gönczy, L., Varró, D., 2008. Formal analysis of BPEL workflows with compensation by model checking. International Journal of Computer Systems and Engineering 23.

Lapadula, A., Pugliese, R., Tiezzi, F., 2008. A formal account of WS-BPEL, in: Proceedings of the 10th International Conference on Coordination Models and Languages, Springer-Verlag, Berlin, Heidelberg. pp. 199–215.

Lertphumpanya, T., Senivongse, T., 2008. Basis path test suite and testing process for WS-BPEL. WSEAS Transactions on Computers 7, 483–496.

Li, Z.J., Tan, H.F., Liu, H.H., Zhu, J., Mitsumori, N.M., 2008. Business-process-driven gray-box SOA testing. IBM Systems Journal 47, 457–472.

Liu, H., Li, Z., Zhu, J., Tan, H., 2007. Business process regression testing, in: Proceedings of the 5th International Conference on Service-Oriented Computing, Springer-Verlag, Berlin, Heidelberg. pp. 157–168.

Lohmann, N., 2008. A feature-complete Petri net semantics for WS-BPEL 2.0, in: Proceedings of the 4th International Conference on Web Services and Formal Methods, Springer-Verlag, Berlin, Heidelberg. pp. 77–91.

Luebke, D., et al., 2012. BPELUnit. http://bpelunit.net.

Morimoto, S., 2008. A survey of formal verification for business process modeling, in: ICCS '08: Proceedings of the 8th International Conference on Computational Science, Part II, Springer-Verlag, Berlin, Heidelberg. pp. 514–522.

Mustafa Bozkurt, M.H., Hassoun, Y., 2010. Testing Web Services: A Survey. Technical Report TR-10-01. Department of Computer Science, King's College London.

Myers, G.J., 2004. The Art of Software Testing. John Wiley & Sons. second edition.

Nimmer, J.W., Ernst, M.D., 2001. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. Electronic Notes in Theoretical Computer Science 55, 255 – 276. RV'2001, Runtime Verification (in connection with CAV '01).

Nimmer, J.W., Ernst, M.D., 2002. Automatic generation of program specifications. SIGSOFT Software Engineering Notes 27, 229–239.

OASIS, 2007. WS-BPEL 2.0 standard. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

O'Callahan, R., Jackson, D., 1997. Lackwit: A program understanding tool based on type inference, in: Proceedings of the 1997 International Conference on Software Engineering, ACM Press. pp. 338–348.

Oracle, 2012. Oracle BPEL Process Manager. `http://www.oracle.com/technetwork/middleware/bpel`.

Palomo-Duarte, M., García-Domínguez, A., Medina-Bulo, I., 2009. Enhancing WS-BPEL dynamic invariant generation using XML Schema and XPath information, in: Gaedke, M., Grossniklaus, M., Díaz, O. (Eds.), ICWE, Springer. pp. 469–472.

Ratcliff, S., White, D.R., Clark, J.A., 2011. Searching for invariants using genetic programming and mutation testing, in: Proceedings of the 13th annual conference on Genetic and evolutionary computation, ACM, New York, NY, USA. pp. 1907–1914.

Rusli, H.M., Ibrahim, S., Puteh, M., 2011. Testing web services composition: A mapping study. Communications of the IBIMA 2011, 12. DOI: 10.5171/2011.598357.

Saraswathi, R., Singh, J., 2013. Oracle SOA BPEL Process Manager 11gR1 – A Hands-on Tutorial. Professional expertise distilled, Packt Publishing.

SPI&FM Group, 2012. Official Takuan home site. `http://neptuno.uca.es/~takuan`.

UCASE Software Engineering Group, . WS-BPEL composition repository. `http://neptuno.uca.es/redmine/projects/show/wsbpel-comp-repo`.

UCASE Software Engineering Group, 2012. Fork of the ActiveBPEL 4.1 WS-BPEL and BPEL4WS engine. `https://neptuno.uca.es/redmine/projects/activebpel`.

W3C, 2007. XSL Recommendations (including XPath 1.0 and 2.0). `http://www.w3.org/Style/XSL`.

Zakaria, Z., Atan, R., Ghani, A.A.A., Sani, N.F.M., 2009. Unit testing approaches for BPEL: A systematic review. Asia-Pacific Software Engineering Conference 0, 316–322.

Zheng, Y., Zhou, J., Krause, P., 2007. An automatic test case generation framework for web services. Journal of Software 2, 64–77.