AN ARCHITECTURE FOR DYNAMIC INVARIANT GENERATION IN WS-BPEL WEB SERVICE COMPOSITIONS

M. Palomo Duarte, A. Garca Domnguez and I. Medina Bulo
Department of Computer Languages and Systems, E.S.I., University of Cádiz
C/ Chile s/n. 11002 Cádiz, Spain

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

Abstract: Web services related technologies (especially web services compositions) play now a key role in e-Business and its future. Languages to compose web services, such as the OASIS WS-BPEL 2.0 standard, open a vast new field for programming in the large. But they also present a challenge for traditional white-box testing, due to the inclusion of specific instructions for concurrency, fault compensation or dynamic service discovery and invocation. Automatic invariant generation has proved to be a successful white-box testing-based technique to test and improve the quality of traditional imperative programs. This paper proposes a new architecture to create a framework that dynamically generates likely invariants from the execution of web services compositions in WS-BPEL to support white-box testing.

1 INTRODUCTION

Web Services (WS) and Service Oriented Architectures (SOA) are, according to many authors, one of the keys to understand e-Business in the early future (Heffner and Fulton, 2007). But as isolated services by themselves are not usually what customers need, languages to program in the large composing WS into more complex ones, like the OASIS standard WS-BPEL 2.0 (OASIS, 2007), are becoming more and more important for e-Business providers (Curbera et al., 2003).

There are two approaches to program testing (Bertolino and Marchetti, 2005): black-box testing is only concerned about program inputs and outputs, while white-box testing takes into account the internal logic of the program. The latter produces more refined results, but it requires access to the source code. However, WS-BPEL presents a challenge for traditional white-box testing, due to the inclusion of WS-specific instructions to handle concurrency, fault compensation or dynamic service discovery and invocation (Bucchiarone et al., 2007).

Automatic invariant generation (Ernst et al., 2001) has proved to be a successful technique to assist in white-box testing of programs written in traditional imperative languages. Let us note that, throughout this work, to be consistent with related work (Ernst et al., 2001) we use *invariant* and *likely invariant* in its broadest sense: properties which hold always or in a specified test suite at a certain program point, respectively. We consider that dynamically generating such invariants, backed by a good test suite, can become an interesting help in WS-BPEL white-box testing.

We propose a new architecture for a framework to dynamically generate likely invariants of a WS-BPEL composition from actual execution logs of a test suite running on a WS-BPEL engine. These invariants can be an interesting aid for white-box testing the composition.

The rest of the paper is organized as follows. First, we explain the particularities of WS compositions built with WS-BPEL. The next section shows how dynamic generation of invariants can succesfully help in WS composition white-box testing. In the following main section we introduce our proposed architecture, discuss how to solve some technical problems that might arise during its implementation, and show an example of the expected results. Finally, we compare our proposal with other alternatives and present some conclusions and an outline of our future work.

Palomo Duarte M., Garca Dominguez A. and Medina Bulo I. (2008).
AN ARCHITECTURE FOR DYNAMIC INVARIANT GENERATION IN WS-BPEL WEB SERVICE COMPOSITIONS.
In *Proceedings of the International Conference on e-Business*, pages 37-44
DOI: 10.5220/0001908800370044

2 WS COMPOSITIONS AND WS-BPEL

The need to compose several WS to offer higher level and more complex services to suit customers requirements was detected and satisfied by the leading companies in the IT industry with the non-standard specification BPEL4WS, which was submitted to OASIS in 2003 (OASIS, 2003). Soon after, OASIS created the *WS Business Process Execution Language Technical Committee* to work in its standardization, releasing the first standard specification, WS-BPEL 2.0, in 2007 (OASIS, 2007).

Standardization has been an important milestone for WS-BPEL wide adoption by SOA leading tools, being a key interoperability feature offered by many e-Business systems (Domnguez Jimnez et al., 2007). This way, companies providing services can take advantage of the new possibilities that this programming-in-the-large technology allows for: concurrency, fault recovery and compensation or dynamic composition using loosely coupled services from different providers selected through several criteria (such as cost, reliability or response time).

WS-BPEL is an XML-based language using XML Schema as its type system. WS-BPEL specifies service composition logic through XML tags defining activities like assignments, loops, message passing or synchronization. It is independent of the implementation and platform of both the service composition system and the different services used in it. There are other W3C-standardized XML-related technologies (W3C, 2008) at its foundation:

XPath. allows us to query XML documents in a flexible and concise way. Although the latest version of the language is XPath 2.0, WS-BPEL uses by default XPath 1.0.

SOAP. is an information exchange protocol commonly used in WS. It is platform independent, flexible and easy to extend.

WSDL. is a language for WS interface description, detailing the structure every message to be exchanged, the interfaces and locations of the services offered, their bindings to specific protocols, etc.

There are several technologies that can extend WS-BPEL. They are usually referred to as the *WS-Stack* (Papazoglou, 2007). One of the most interesting is UDDI, a protocol that allows maintaining repositories of WSDL specifications, easing the dynamic discovery and invocation of WS and the access to their specifications. UDDI 3.0 is an OASIS Standard (OASIS, 2008). But the inherent dynamic nature of these technologies also implies new challenges for program testing, as most traditional white-box testing methodologies cannot be directly applied to this language.

3 WS-BPEL WHITE-BOX TESTING

WS composition testing is one of the challenges for its full adoption in industry in the forthcoming years. The dynamic nature of WS poses a challenge for testing, having to cope with aspects like run-time discovery and invocation of new services, concurrency and fault compensation.

Little research has been done on applying white-box testing directly on WS-BPEL code. Main approaches (Bucchiarone et al., 2007) create simulation models in testing-oriented environments. But simulating a WS-BPEL engine is very complex, as there is a wide array of non-trivial features to be implemented, such as fault compensation, concurrency or event handlers. In addition, it is sometimes necessary to translate the code of the WS-BPEL composition to a second language to check its internal logic.

In case any of these features is not properly implemented, compositions would not be accurately tested. So we consider that this is an error-prone process, as it is not based on the actual execution of the WS-BPEL code in a real environment, that is a WS-BPEL engine invoking actual services.

Therefore, we propose using dynamically generated invariants from actual execution traces as a more suitable approach.

3.1 Using Invariants

An invariant is a property that holds at a certain point in a program. Classical examples are function pre-conditions and post-conditions, that is, assertions which hold at the beginning and end of a sequence of statements. There are also loop invariants, which are properties that hold before every iteration and after the last one.

Let us illustrate these concepts with a simple example. Suppose that we want to sum all the integers from 0 to a given natural n. In this case, we could define this simple algorithm:

```
1. $r \leftarrow 0$
2. From $i \leftarrow 1$ to $n$:
     $r \leftarrow i + r$
3. Return $r$
```

This algorithm has the pre-condition $n > 0$ in step 1, as n is natural by definition. Looking at the loop in step 2, we can tell that the loop invariant $r = \sum_{j=0}^{i-1} j$ holds before every iteration. Since in step 3 we have exited the loop, we will have $i = n + 1$, which we can substitute in the previous loop invariant to formulate the post-condition for step 3 and the whole algorithm, $r = \sum_{j=0}^{n} j$. From this post-condition we can tell that the algorithm really does what it is intended to do.

Manually generated invariants have been successfully used as above to prove the correctness of many popular algorithms to this day. Nonetheless, their generation can be automated. Automatic invariant generation has proved to be a successful technique to test and improve programs written in traditional structured and object-oriented programming languages (Ernst et al., 2001).

Invariants generated from a program have many applications:

Debugging. An unexpected invariant can highlight a bug in the code which otherwise might have been missed altogether. This includes, for instance, function calls with invalid or unexpected parameter values.

Program Upgrade Support. Invariants can help developers while upgrading a program. After checking which invariants should hold in the next version of the program and which should not, they could compare the invariants of the new version with those of the original one. Any unexpected difference would indicate that a new bug had been introduced.

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

Verification. We can compare the specification of the program with the actual invariants obtained to see if they satisfy.

Test Suite Improvement. A wrong likely invariant dynamically generated, as we'll see in next section, can demonstrate a deficiency in the test suite used to infer it.

3.2 Automatic Invariant Generation

Basically, we there are two approaches when generating invariants automatically: static and dynamic.

Static invariant generators (Bjørner et al., 1997) are most common: invariants are deduced statically, that is, without running the program. To deduce invariants, its source code is analyzed (specially data and control flows). On one hand, invariants generated this way are always correct. But, on the other hand, its number and quality is limited due to the inner limitations of the formal machinery which analyzes the code, specially in unusual languages like WS-BPEL.

Conversely, a dynamic invariant generator (Ernst et al., 2001) is a system that reports likely program invariants observed on a set of execution log files. It includes formal machinery to analyze the information in the logs about the values held by variables at different program points, such as the entry and exit points for functions or loops.

The process to generate dynamic invariants is divided into three main steps:

1. An *instrumentation step* where the original program is set up so that, during the later execution step, it generates the execution log files. This step is called instrumentation step because the usual way to do it is by adding, at the desired program points, logging instructions. These instructions write to a file the name and value of the variables that we want to observe at those points and otherwise have no effect on the control and data flow of the process. Sometimes it is also necessary to modify the environment where the program is going to be executed.
2. An *execution step* in which the instrumented program will be executed under a test suite. During each test case an execution log is generated with all the necessary data and program flow information for later processing.
3. An *analysis step* where formal methods techniques are applied to obtain invariants of the variables logged at the different program points.

Thus, the dynamic generation of invariants does not analyze the code, but a set of samples of the values held by variables in certain points of the program. Wrong invariants do not necessarily mean bugs in the tested program, but rather they might come from an incomplete test suite. If the input x is a signed integer and we only used positive values as test inputs, we will probably obtain the invariant $x > 0$ at some program point. Upon inspection, we would notice that invariant and improve our test suite to including cases with $x < 0$.

3.3 Dynamic Invariant Generation in WS-BPEL Compositions

We consider the dynamic generation of invariants to be a suitable technique to support WS-BPEL composition white-box testing. If we use a good test suite, all of the complex internal logic of our BPEL composition (compensation, dynamic discovery of services, etc.) will be reflected in the log files of the different executions, and the generator will infer significant invariants.

Generally, due to its dynamic nature, the more logs we provide the generator, the better results it will produce. In case we obtain seemingly false invariants in a first run, we will be able to certify if they were due to an incomplete test suite or to actual bugs in a second run with an improved test suite including additional suitable test cases.

Another important benefit is that all the information in the logs is collected from direct execution of the composition code, using no intermediate language of any sort. This way we avoid errors that could arise in any translation of the WS-BPEL code or the simulation of the real-world environment, that is, the WS-BPEL engine and invoked services.

An important problem to solve is that usually, all external services will not be available for testing, due to access restrictions, reliability issues or resource constraints. Or it could also just be that we wanted to define several what-if scenarios with specific responses from several external WS. Thus, we will also have to allow for replacing some external services with mockups, that is, dummy services which will reply our requests with predefined messages.

4 PROPOSED ARCHITECTURE

An outline of our proposed architecture for a WS-BPEL dynamic invariant generator is shown in the figure 1.

We are using a classical pipeline-based architecture which has, in general terms, three coarse-grained steps corresponding to the three general steps of the dynamic invariant generation process which we described above.

We detail further their WS-BPEL specific issues below.

4.1 Instrumentation Step

In this is the step where we take the original program and perform the necessary changes on it in order to produce the information that the invariant generator needs.

In our case, we will take the original WS-BPEL process composition specification with its dependencies and automatically instrument it. If necessary we will create any additional files needed for its execution in the specific WS-BPEL engine which we will be using.

Additional logic to generate the logs that we need can be added in three ways:

1. Firstly, we could modify the execution environment itself. We could use an existing open-source WS-BPEL engine and modify it in order to produce the log files that we need for any process executed in it.

   The degree of effort involved would depend on the logging capabilities already implemented in the engine. Most engines include facilities to track process execution flow, but tracking variable values is not widely implemented. Of course, modifying the code would take a considerable effort.

2. Secondly, we could modify the source code of the WS-BPEL composition to be tested, adding calls to certain logging instructions at the desired program points. These instructions would not change the behavior of the process, being limited to transparently inspect and log variable values.

   In a similar way to the previous method, we might be able to use any existing engine-specific WS-BPEL logging extension. It could append messages to a log file, access a database or invoke an external logging web service, for instance.

   Using this approach we would have to instrument the two different languages used by the WS-BPEL standard: the WS-BPEL language itself, which is XML-based, and the XPath language that is used to construct complex expressions for conditions, assignments, and so on.

3. And finally, we could implement our own logging XPath extension functions.

   These new functions would be called from an instrumented version of the original WS-BPEL composition source code, and included in external modules which are reasonably easy to automatically plug into most WS-BPEL engines. This is a hybrid approach, as both the composition and the engine need to be modified.

   It is quite likely that using both internal (that is, inside the WS-BPEL engine) and external modifications (using new XPath extension functions) will allow us to obtain more detailed logs with less effort than any of the previous approaches.

We also take into account the fact that there are many WS-BPEL engines currently available. Engine-specific files for the deployment of the composition under the selected engine may have to be generated automatically on the fly, to abstract the user from the technical details involved.

4.2 Execution Step

In this step, we will take the previously instrumented program and run it under a test suite to obtain the logs

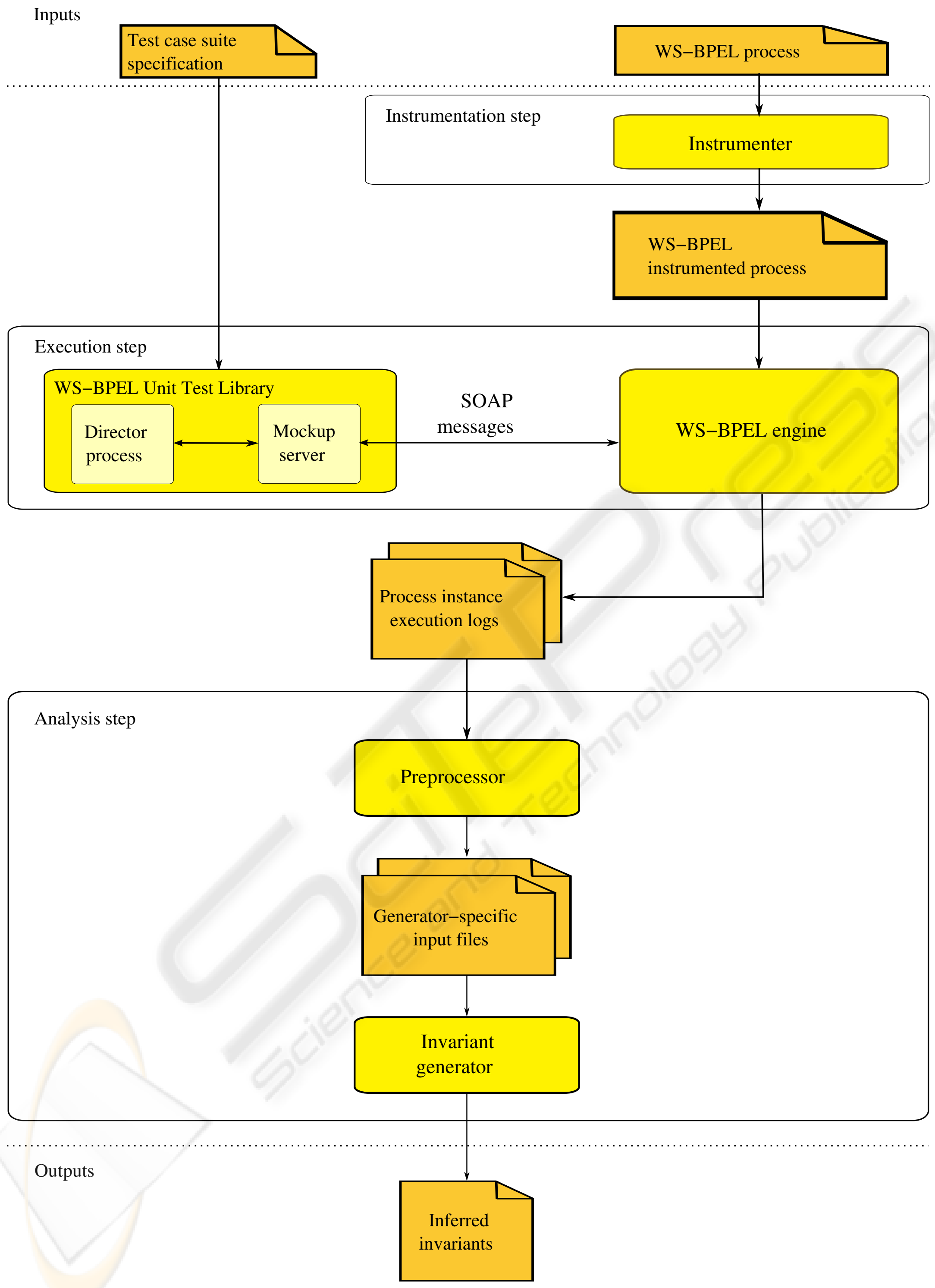

Figure 1: Proposed architecture for dynamic invariant generation.

and other information that we need for the next step. Specifically, we will take our instrumented WS-BPEL composition and run it under each test case detailed in the test suite specification. Each of them will define the initial input message that will cause a new instance of the WS-BPEL process to be started, as well as the outputs of the external services required by the WS-BPEL process that we wish to model as mockups.

The ability to model none, some or even all of the external services as mockups will enable us to obtain invariants reflecting different situations. On one hand, if we do not use mockups at all, but only invoke actual services, we will be studying the complete WS-BPEL composition in the real-world environment. On the other, if every external service is replaced with a mockup with predefined responses, we will be focusing on the internal logic of the composition itself and how it behaves in certain scenarios. We can also settle for a middle point in a hybrid approach.

After this step is done, every test case will have generated its own execution log, which we will pass on to the next step. We will need roughly two components to make this possible:

- A WS-BPEL engine for running the process itself, which will invoke the external services as needed. Making the engine use a mockup for an external service could be achieved in basically two ways: by modifying the service address included in the WSDL source files, or by creating (or modifying) the engine-specific files with the proper values.
- A WS-BPEL unit test library which will deploy and act both as a client, invoking our composition with the desired parameters, and as the external mockup services for the WS-BPEL process. These services will behave according to the external test case specification described above.

This unit test library has to be quite more complex than similar libraries for other languages. It can be divided once more into the following subcomponents:

- A *director* which will prepare and monitor the execution of the whole test suite according to each test case specification. Ideally, it should also be able to deploy and undeploy the WS-BPEL process from our selected engine.
- A *mockup server*, properly set up by the director, which will handle incoming requests and act as the external mockup services required by the WS-BPEL process that we choose to model.

  Mockups have no internal logic of their own, being limited to either replying with a predefined XML SOAP message or failing as indicated in the test case specification.

  There are several lightweight Java-based web servers available for this role, but, if necessary, we believe it would also be feasible to develop it from scratch, being a simple URL → SOAP message matching system.

4.3 Analysis Step

After all the test cases have been executed and logs have been collected, it would seem at first glance to be a matter of just handing them to our invariant generator.

However, it will not be so simple in most cases, because the invariant generator could require certain additional information about the data to analyze to work properly. All information would have to be reformatted according to the input format expected by the invariant generator. This reformatting could range from a simple translation to a thorough transformation of the XML data structures to those available to the invariant generator.

The invariant generator may even accept not only logs, but also a list of constraints already known, and which thus do not need to be generated again as invariants. This would reduce its output size and make it easier to understand for its users. To generate this constraint list, we would have to analyze the XML Schema files contained in the WS-BPEL process and all of its dependencies.

All these obstacles can be overcome through a preprocessor. It could even call the invariant generator with the generated files to finally obtain the invariants of the WS-BPEL composition.

Depending on the number and complexity of the invariants produced by the generator, we could even need to pass them later to a simplifier. It is a program based on formal methods that receives a set of invariants and removes those logically inferred by others.

4.4 Example

We comment briefly an example of the invariants we could infer in the classical WS-BPEL example of the *Loan Approval Service* included in the WS-BPEL 2.0 specification (OASIS, 2007).

This WS-BPEL composition receives loan requests from costumers. Each request includes the amount and certain personal information. The WS-BPEL composition simply notifies the costumer whether his loan request has been approved or rejected. The approval of the loan is based on the amount requested and the risk that a risk assessment WS determines for the costumer according to his personal information. If the amount is below $10,000 and the risk assessment WS considers the applicant a low-risk costumer the loan is automatically approved.

In case the amount is below the threshold but risk is considered medium or high, the composition invokes an external loan approval WS, and its answer is passed to the costumer as the response of the composition. Finally, in case the requested amount is over the threshold no risk checking is done, and the answer of the composition is also that of the external loan approval WS.

Our architecture could infer the following invariants for this example, it were backed by an exhaustive and high-quality test suite:

$$\$request.amount < 10000 \wedge \$risk.level = \text{'low'} \implies \$approval.accept = \text{'yes'}$$

$$\$request.amount < 10000 \wedge \$risk.level \neq \text{'low'} \implies \$approval.accept = response(approver)$$

$$\$request.amount \geq 10000 \implies \$approval.accept = response(approver)$$

$$invoke(customer) = \$approval$$

We can clearly see that the system could infer that approval depends on the amount requested and the response provided by the approver when invoked. The system could also detect that the response we provide the costumer is always the value of the variable *approval*.

Of course, to get results this fine-grained we will need a good test suite. For our example, the test suite would have to contain test cases with amounts over and under the threshold (including the limit values $9,999, $10,000 and $10,001). Personal information causing the risk assessment WS answers both affirmatively and negatively must also be provided, specially for those under the threshold. At any rate, as discussed before, in case we do not obtain the desired invariants in the first run, we could extend the test suite with more test cases refining the invariants obtained. This way, in the next run, we would obtain more accurate invariants.

5 RELATED WORK

In this section we present some related works. There is a wide variety of topics related to our architecture, mainly dynamic invariant generation, WS composition testing and test case generation:
An interesting proposal to use dynamically invariants for WS quality testing is (SeCSE, 2007). It collects several invocations and replies from a WS-BPEL composition to an external WS and dynamically generates likely invariants to check its *Service Level Agreement*. Therefore, it constitutes a black-box testing technique. In contrast, our architecture follows a white-box testing approach, oriented to the generation of the invariants from the internal logic of a WS-BPEL composition.

The relation between the test cases used for dynamically generating invariants and the quality of the invariants derived is studied in (Gupta, 2003). Augmenting a test suite with suitable test cases can be an interesting way to increase the accuracy of the invariants inferred by our architecture.

Dynamo (Baresi and Guinea, 2005) is a proxy-based system to monitor if a WS-BPEL composition holds several restrictions during its execution. We think it might be useful as a way to check if the invariants obtained from our architecture hold while it is running in a real-world environment.

Test cases for a WS-BPEL composition are automatically generated in (Zheng et al., 2007) according to state and transition coverage criteria. We could assure the quality of the invariants generated by using them as inputs for our architecture.

6 CONCLUSIONS AND FUTURE WORK

The fact that WS are the future of e-Business is already a given, thanks to their platform independence and the abstractions that they provide. The need to orchestrate them to provide more advanced services suiting costumer's requirement has been satisfied through the WS-BPEL 2.0 standard.

However, WS-BPEL compositions are difficult to test, since traditional white-box testing techniques are difficult to apply to them. This is because of the unusual mix of features present in WS-BPEL, such as concurrency support, event handling or fault compensation. We have showed how dynamically generated likely invariants backed by a good test suite can become a suitable and successful help to solve these difficulties, thanks to their being based on actual execution logs.

In this work, we have proposed a pipelined architecture for dynamically generating invariants from a WS-BPEL composition. Requirements on every of the components have been identified, leaving as our next future work finding suitable systems for each one.
Once the architecture is completely implemented we will perform an experimental evaluation of the framework under several compositions. This way we will test its reliability and evaluate its results through metrics such as quality of the invariants generated or time

taken to infer them.

Looking further ahead, we will later study the relation between the quality of the invariants generated and the test case suite used to infer them. For this we can use different WS-BPEL compositions with their specifications and different test suites providing certain coverage criteria (branch coverage, statement coverage, . . .) of them.

Finally, we could use the invariants generated by our proposal to support WS-BPEL white-box testing and check if it improves its results.

ACKNOWLEDGEMENTS

This work has been financed by the *Programa Nacional de I+D+I* of the Spanish *Ministerio de Educacin y Ciencia* and *FEDER* funds through SOAQSim project (TIN2007-67843-C06-04).

REFERENCES

Baresi, L. and Guinea, S. (2005). Dynamo: Dynamic monitoring of WS-BPEL processes. In Benatallah, B., Casati, F., and Traverso, P., editors, *ICSOC*, volume 3826, pages 478–483. Lecture Notes in Computer Science, Springer.

Bertolino, A. and Marchetti, E. (2005). A brief essay on software testing. In Thayer, R. H. and Christensen, M., editors, *Software Engineering, The Development Process*. Wiley-IEEE Computer Society Pr, 3 edition.

Bjørner, N., Browne, A., and Manna, Z. (1997). Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1):49–87.

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

Curbera, F., Khalaf, R., Mukhi, N., Tai, S., and Weerawarana, S. (2003). The next step in Web Services. *Communications of the ACM*, 46(10):29–34.

Domnguez Jimnez, J. J., Estero Botaro, A., Medina Bulo, I., Palomo Duarte, M., and Palomo Lozano, F. (2007). El reto de los servicios web para el software libre. In *Proceedings of the FLOSS International Conference 2007*, pages 117–132, Jerez de la Frontera. Servicio de Publicaciones de la Universidad de Cdiz.

Ernst, M. D., Cockrell, J., Griswold, W. G., and Notkin, D. (2001). Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123.

Gupta, N. (2003). Generating test data for dynamically discovering likely program invariants. In *ICSE, Workshop on Dynamic Analysis*.

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

OASIS (2003). OASIS members form web services business process execution language (WS-BPEL) technical committee. http://www.oasis-open.org/news/oasis_news_04_29_03.php.

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

OASIS (2008). OASIS standards. http://www.oasis-open.org/specs/.

Papazoglou, M. (2007). Web services technologies and standards. computing surveys (enviado para revisin).

SeCSE (2007). A1.D3.3: Testing method definition V3. http://secse.eng.it/wp-content/uploads/2007/08/a1d33-testing-method-definition-v3.pdf.

W3C (2008). W3C technical reports and publications. http://www.w3.org/TR/.

Zheng, Y., Zhou, J., and Krause, P. (2007). An automatic test case generation framework for web services. *Journal of Software*, 2(3):64–77.