JOpera: an Agile Environment for Web Service Composition with Visual Unit Testing and Refactoring

Cesare Pautasso

Department of Computer Science, Swiss Federal Institute of Technology (ETHZ), ETH Zentrum, 8092 Zürich, Switzerland

pautasso@inf.ethz.ch

Abstract

Agile methodologies employ light-weight development practices emphasizing a test-driven approach to the development of software systems. Modern agile development environments support this approach by providing tools that automate most of the work required to effectively deal with change, including unit testing and different forms of refactoring. In this paper we discuss how to apply such techniques within the JOpera Visual Composition Language. More precisely, we show how we used the visual language to implement a regression testing framework for compositions written in the language itself, and how we introduced support in the visual environment for refactorings such as renaming, synchronization of service interface changes, and extraction/inlining across different levels of nesting. This is done in the context of the Web service composition tools provided with the JOpera for Eclipse *research platform.*

1 Introduction

Agile methods focus on embracing change as part of the ordinary software development practice [1]. Test-driven development, refactoring, and immediate feedback are all important aspects of agile methodologies for which modern development tools provide more and more support in the context of traditional (and textual) programming languages.

Dealing with change is also an important issue in the context of Web service composition, due to the brittleness of the distributed systems built out of the composition of remote Web services. Not only compositions are sensitive to the run-time availability of their component services, but also service interfaces may evolve independently of the compositions including them. Refactoring plays a major role to address the latter, as far as a composition can be automatically synchronized with the changing interface of its component services.

Furthermore, change may also affect the implementation of the services: a Web service may be upgraded without modifying its interface so that its clients continue to work unaffected. After such change occurs, it is important to be able to automatically ensure that a composition may continue to rely on the modified service. Thus, the ability to automatically determine the impact of a modified service on a composition that depends on it becomes an important feature of an agile composition environment.

By proposing to apply agile methodologies to visual Web service composition, this paper makes the following important contributions. First of all, we present several kinds of visual refactorings that can be automatically applied to a composition in order to restructure it ensuring that it remains consistent. Second, we describe how to use a visual composition language to implement a regression testing framework for compositions written in the language itself [4]. This is a significant improvement with respect to many current Web service composition languages, due to their limited reflection capabilities. These advanced features have been implemented in the latest version of the JOpera visual composition environment, which has been recently ported to the Eclipse platform using the Graphical Editing Framework. JOpera for Eclipse can be downloaded from [3].

2 Refactoring a Web service composition

JOpera supports several kinds of refactoring [2]: renaming, extraction and inlining of sub-compositions, as well as the semi-automatic synchronization of service interface changes.

2.1 Renaming

Renaming is provided implicitly by the editing tools that always ensure the consistency of the composition model after one element has been renamed. Thus, all references to a renamed element are updated transparently.

As a generalization of this refactoring, the elements of a composition can also be *moved* across different locations (e.g., physical source files or logical packages) without breaking the references to them found in other elements. This way, the structure of a composition can be reorganized without having to manually update all of its elements.

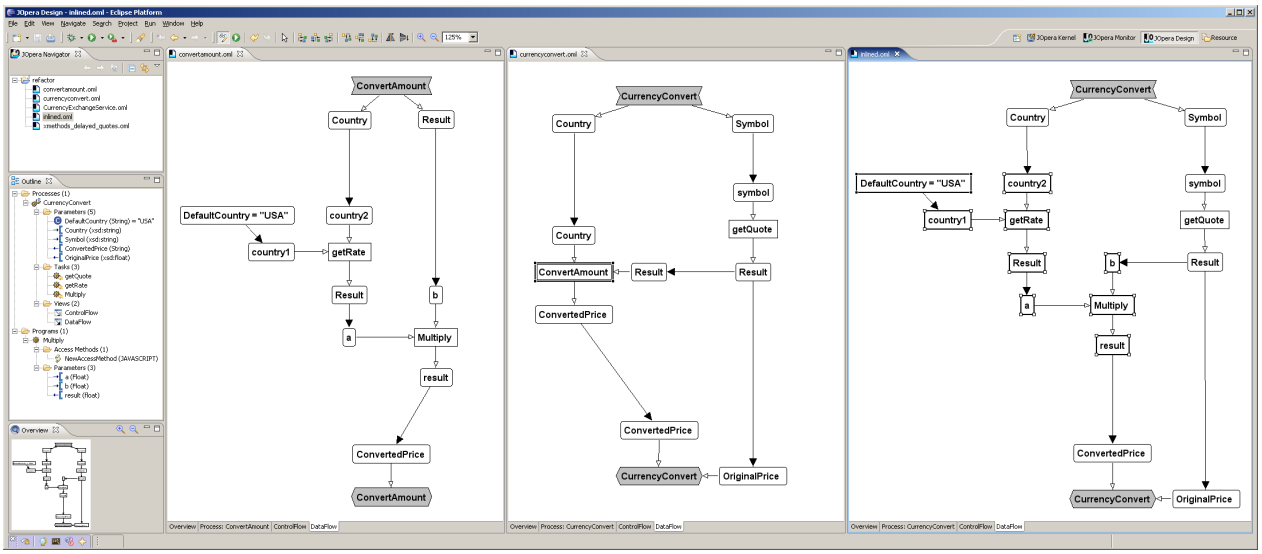

Figure 1. Screenshot of JOpera for Eclipse showing the original composition to be inlined (left) into another one (center) and the result of the refactoring while it is being executed (right)

2.2 Extraction and Inlining

Extraction and Inlining take a prominent role due to the nesting capabilities of the composition language. Since compositions of services can be recursively reused as a service, providing support for easily shifting elements of a composition along the nesting hierarchy is an important characteristic of an agile composition environment. JOpera offers two refactoring wizards, that can be applied to the currently selected services of a composition. One wizard collapses (or extracts) the selection into a new sub-composition; the inverse expands (or inlines) the content of the currently selected sub-composition. An example is shown in Figure 1.

Before applying the changes, the refactoring algorithm used by the wizards checks the selection to validate it against a set of pre-conditions, since not all possible combinations of services can be extracted into a valid sub-composition. Thus, the developer is warned in advance if it is not possible to apply the refactoring. First of all, the control flow between the services is checked. For example, the services are ranked in the control flow graph according to their dependencies. With this information, the refactoring algorithm verifies that not only the selected sub-graph is connected, but also that all paths from the low-ranking services to the high-ranking ones lead over a service selected to be extracted. The data flow graph is also checked to define the input and output parameters to be associated with the extracted sub-composition. More precisely, the input parameters of the new sub-composition reflect the source parameters of the data flow edges that are inbound into the selection. Symmetrically, the output parameters are cloned from the destination parameters which are targeted by outgoing edges.

In addition to preserving the topology of the control and data flow graph of the refactored composition, this refactoring must also ensure that the visual layout of the extracted (or inlined) elements is maintained, so that the developer's mental map of the flow remains consistent.

2.3 Synchronizing Interface Changes

In the context of Web service composition, the ability to modify the interface of a service without breaking a composition is very important. This situation is quite likely to happen, especially if Web services are composed by organizations different than the ones publishing them. Furthermore, even if the services and their compositions are co-developed within the same organization, a few iterations of the publish-integrate cycle may be required before the interfaces of the services stabilize. By using this refactoring, services must not be recomposed from scratch once their interface is updated.

In JOpera this refactoring takes two forms depending on whether the service definitions are tightly or loosely coupled with the composition.

In the first case, direct modifications to the signature of a service stored in JOpera's internal model will be immediately reflected in the composition. For example, if a parameter is removed from a service description, the corresponding parameter in all of the compositions using the service will be immediately removed along with its data flow edges. Likewise, if a new parameter is added to a service description, the new parameter will be immediately be available to be connected with other ones.

If service interfaces are modified outside of the control of JOpera, it should still be possible to synchronize such interfaces changes by minimizing the impact on the existing

compositions. To do so, JOpera allows developers to import the new version of the interface description and to automatically refactor one or more compositions to use the new version. The refactoring algorithm determines the commonalities between the two versions of the service by matching the names and data types of the parameters so that existing data flow connections can be maintained. Developers greatly appreciate the ability not to have to reconnect the modified services as after applying this refactoring they only have to manually connect the parameters that could not be automatically matched.

3 Testing a Web service composition

Testing a composition involves several aspects, for which an agile environment should provide support using the appropriate visual tools.

JOpera keeps a clear separation between the model of a composition and the description of its component services. This makes it possible to test each independently. A composition may be bound to stubs (i.e., testing-version of the services) so that its execution for testing purposes will not perturb the "real" Web services that will eventually be invoked when the composition is deployed in production mode. Similarly, as a form of acceptance test, with JOpera, tools are provided to quickly build a Web service client (i.e., a one-service composition) that can be used to interactively test a service, in order to verify assumptions about the provided functionality.

In this paper we focus on testing compositions as a whole, in order to guarantee that despite changes of the implementation of one or more services, the overall behavior of a composition remains unaffected. To address this problem, we follow an approach that leverages the persistent execution capabilities of JOpera, where snapshots of the complete state of the execution of a composition can be taken, made persistent and compared with existing snapshots.

The JOpera regression testing framework has been implemented as a visual composition of three different services (Figure 2). First, a registry is queried to discover the available test cases. Following JUnit's convention, the `LookupProcesses` task retrieves all compositions whose name stems with the word `test`. These are invoked using dynamic late binding (`Call`). Finally a snapshot of their execution state is taken and compared with the expected one (`SnapshotAndCompare`).

If mismatches are detected, the corresponding errors are reported in the `Result` parameter and gathered for all tests in the `TestResult` parameter. Since such snapshots include the values of all data flow parameters of the composition, developers can immediately pinpoint the cause of a failed test case, as JOpera highlights such parameters in the data flow graph of the composition.

If no expected snapshot can be found at the given `OraclePath` repository, a new snapshot using the current state will be created. This way it becomes easier to define

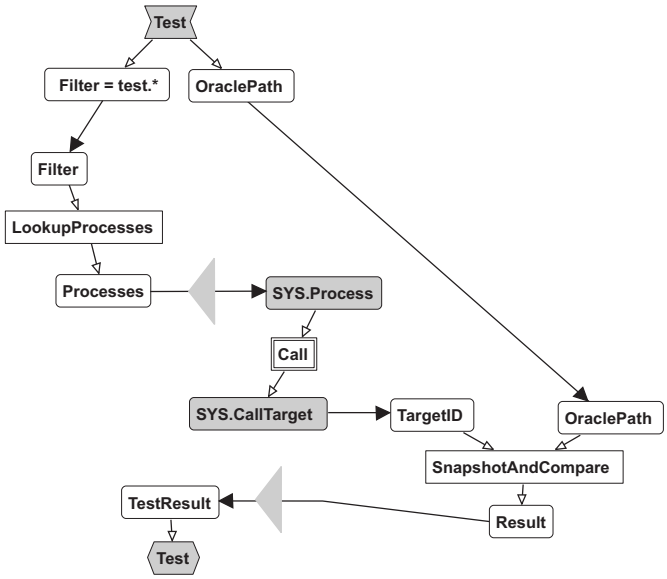

Figure 2. Data Flow of the Regression Testing Framework written in the JOpera Visual Composition Language

the expected state of a composition, as it is automatically recorded. To ensure its correctness, manual inspection of the snapshot is required once, after the test has been run for the first time, while the snapshot can be reused for all subsequent runs.

We are currently extending this regression testing framework along two directions. 1) It should be possible to compare only a subset of the entire execution state of a composition in order to disregard test failures due to non-deterministic parameter values (e.g., time-dependent data). 2) Developers may also proactively annotate the data flow of a composition with assertions to be verified during each execution of a test case. This would provide an additional level of protection with respect to the one provided by the state comparison.

Acknowledgements Part of this work is supported by grants from the *Hasler Foundation* (DISC Project No. 1820).

References

[1] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2nd edition, November 2004.

[2] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[3] C. Pautasso. JOpera: Process Support for more than Web services. `http://www.jopera.org`.

[4] C. Pautasso and G. Alonso. The JOpera Visual Composition Language. *Journal of Visual Languages and Computing*, 16(1–2):119–152, 2004.