Содержание

Contents

RITA—a Framework Integration and Testing Application

Juha Taina, Professor Jukka Paakki, Raine Kauppinen

Department of Computer Science, University of Helsinki

P.O. Box 26 (Teollisuuskatu 23)
FIN-00014 University of Helsinki, Finland

E-mail: {Juha.Taina, Jukka.Paakki,
Raine.Kauppinen}@cs.Helsinki.FI

Abstract

In this paper we give the first introduction of our product line testing application RITA. RITA is designed for framework and framelet based application testing. It includes services for interface class identification, code profiling, coverage criteria analysis, driver and stub generation, test management, and statistics.

Contents

1 Introduction

A product line approach in software development is based on the fact that usually a software development company creates several closely related software products at the same time. When those products are treated as a family, equal approaches and common code may be shared between the products.

From a testing point of view, testing a product line member may be seen as a problem of testing an incomplete program. While functionality common to all product line members is present and well tested, application-specific functionality may be missing or only partially tested. This special nature should be taken into account in product line testing.

We have taken a framework-based approach in product line testing. A framework is considered a common basis for all product line members. It is well defined and tested. Next to this, smaller framework-type entities called framelets are used to define functionality that is common to a subset of the product line family. Also framelets are well defined and tested, but their interfaces to other elements of code may need to be tested. Finally, each product line family member has application-specific code that needs full testing. Also regression testing must be taken into account in all phases of testing framework and framelet-based applications.

The special nature of framework and framelet-based software helps to identify potential test issues. The characteristics of such software are as follows:

Test state. Most of the code is already fully or partially tested in frameworks and framelets.

Interfaces. All application-specific code is defined under well-defined interfaces. Thus, only specific pars of frameworks and framelets need extra regression testing.

Automated test alternatives. Since most of the application code is generated automatically, the code structure may be used to identify test cases.

Using the framework-based testing approach, we are developing our product line testing tool RITA (fRamework Integration and Testing Application). In this paper we present the core functionality of RITA.

The rest of the paper is organized as follows. In Section 2 we give a brief summary of related work. Section 3 gives the necessary background of framework-based applications. In Section 4 we introduce our framework-testing environment RITA. Finally, Section 5 concludes the paper.

2 Related Work

The product line approach for software development is currently under extensive research. There have been a few large scale projects that have studied the product line approach and product families, for example the ARES project [7]. Also, SEI has developed a framework for software product line practice [12]. In addition, some case studies involving software product lines have been made [1,4].

However, surprisingly little is written about testing in the product line approach. Naturally, the traditional object-oriented methods for testing large applications or frameworks and reusing software components can be used [2,3,5,6,9,13], but there is also growing demand for a well-defined product line testing process and methodology including tool support. One approach to product line testing is presented in SEI's framework for software product line practice [12]. Also McGregor and Sykes have defined a testing process and introduced methodology and tools that can be used in the product line approach [10].

3 Framework Issues

A *framework* can be seen as a meta-model for an object-oriented family of software products. It offers a basic structure that is similar or almost similar to all members of the software product family. As such it fits extremely well *product lines* which are clearly families of similar software products.

When we consider a member of a product line family, the framework defines the central concepts of the product architecture. The framework is used to create a skeleton product that can later be expanded. Thus, not all of the architecture is described in the framework. The unspecified parts are open for application extensions.

The framework-based skeleton product contains a set of *template objects*. A template object, as described in [11], is an interface object from the static framework-created code to dynamic application extension code.

Each template object has a relationship to zero or more *hook objects*. Each hook object offers an entry point that may be extended with application specific code. Template objects offer a static interface to hook objects, and hook objects offer skeleton functionality that is later expanded by application specific code. The hook objects are often called *hot spots* since they offer dynamic interfaces for application code. Similarly, template objects are called *cold spots* since they are static in nature.

The template and hook objects form a relationship, as presented in [11]. The relationships are as follows.

Unification relationship: The template and hook object are the same object. Thus, the template object is a hot spot itself. This is the most common case in framework generators. They generate a set of tasks that must be fulfilled. A finished task is an object or a method, both of which may be considered a combination of a template and a hook object.

Connection relationship: A template object has a dynamic relationship to one or more hook objects. This is a more dynamic alternative than the previous ones since it allows different types of hook objects to be connected to a template object. Also the connection may be dynamically defined at software execution time while in a unification relationship the connection is statically defined at compile time.

Recursive unification relationship: The template and hook objects are the same object and have a relationship to another similarly connected object. This relationship allows dynamic lists of hot spots from a template object.

Recursive connection relationship: The template object has a relationship to zero or more hook objects, and the template object is in a subclass of the hook object. This relationship allows dynamic trees of hot spots from a template object.

The relationships are important since they define how and where the framework functionality can be expanded. From a testing point of view, the template and hook objects define where we should start testing. The framework code is automatically generated and expected to work. The framework software vendor has already tested its output. The dynamic product line family member specific code is prone to errors and needs testing.

The application specific code may be hand-designed and written, or it may be automatically generated with framelets. *A framelet* is a small framework, which does not have the main control loop. Framelets have both interfaces to connect to hot spots, and template and hook objects for other application specific extensions. As such they can be used in product line families to generate functionality that is needed only in a subset of the family members, or expand the functionality offered by the framework itself.

The product line approach can be expanded to product line member testing. First, we have the framework which is considered stable and well-defined. It has been tested indirectly when the framework software vendor has tested its product. The product line framelets are not as stable since they include an interface that needs testing. Yet the framelet itself should be relatively stable. Finally, all hand-designed and written application-specific code needs full testing.

4 RITA

The RITA integration and testing environment consists of testing management elements that offer services for white-box testing in product line environments. The elements are as follows: template and hook object identifier, profiler, coverage analyzer, driver and stub generator, test environment, and statistics generator. We will cover the elements in turn in this section.

4.1 Template and Hook Object Identifier

A new application is first analyzed to identify template and hook objects. This information is used in testing and coverage analysis. The identification process is semi-automatic. The template and hook object identifier first recognizes potential template and hook object candidates, and the end user has to choose the actual objects from the candidate list.

The template and hook object information is important since it allows RITA to recognize the interface objects between frameworks, framelets and application code. Alternatively this information may be hand-written or extracted from the framework and framelet generators.

4.2 Profiler

When a newly added application is being tested, it is first run through a profiler. The profiler preprocesses application code so that RITA can

gather data from the actual code execution. This element allows later RITA elements to analyze the executed code, and also to create graphical presentations of the code.

In the first stage the RITA profiler accepts Java code as input. It modifies code without modifying code functionality. A new RITA class is generated. The class gathers information about executed paths and stores it once the program execution is over.

The RITA profiler will be defined so that it can later be expanded to other programming languages than Java. It accepts all standard Java structures. Only exceptions are forbidden in the first version since exception processing can easily violate the normal execution flow of the program. In the later versions of RITA exception processing will also be under research.

4.3 Coverage Analyzer

The Coverage analyzer is one of the main elements of RITA. It accepts a profiler generated file as input and generates white-box based coverage information from the material. In the first prototype we support code and branch coverage.

A very interesting framework-based coverage criteria which we call *hot spot coverage* defines how much of the expanded hot spot functionality has been covered in the tests. Only those hot spots that have framelet or hand-written expansion code are included in the coverage criteria.

The exact theory of hot spot coverage criteria is beyond the scope of this paper. Here is only an informal presentation of the idea.

The hot spot coverage is based on template and hook objects, and paths from template object entry points to their exit points. The set of paths defined for the coverage may be any subset of possible paths. A set of *independent paths*, as defined by McCabe in [8], is one of the best alternatives. An independent path is usually recursively defined as the shortest path through a flow graph that cannot be linearly combined from other independent paths. Basically it says that the first independent path is the shortest path through the flowgraph, and the following paths are reached by adding one uncovered node or edge to the shortest path and calling it the next independent path.

Let us have a template object T that has a relationship to hook objects $H_1, \ldots, H_n$. We define the *hot spot coverage* over a set of paths P to be

the path coverage over P starting from the interface of T and traversing via connected hooks of $H_1, \ldots, H_n$. A hook is connected if it has been expanded by a framelet or a piece of hand-written code.

4.4 Driver and Stub Generator

From a testing point of view, a designed product line family member is an incomplete program. The framework and perhaps framelets are present while some or all of the application-specific code is unwritten. Due to this RITA offers services for automatic driver and stub generation.

In the first prototype of RITA, drivers are based on template and hook objects. RITA uses the identified templates and hooks to generate drivers. In the prototype version we use only unification and connection relationships. The recursive relationships are much more difficult to analyze both for the driver and test case generation.

Since RITA is an integration tool, it allows framelets and application-specific code to be tested separately from the framework. In these cases identified template objects are separated and drivers are connected to them. Thus, testing does not necessarily start from the framework at all. Next to this, regression test cases are also necessary in order to verify that a modified framelet or hand-written code does not affect earlier tested functionality.

Stub generation is needed since the tested product line family member is not necessarily a complete program. However, writing complete stubs would require semantic knowledge of the methods where the stubs are plugged. In the prototype we do not gather such semantic information. Instead we generate only minimal stubs that allow the tested program to be executed and partially tested.

4.5 Test Management Environment

As any testing tool, RITA offers a complete test management environment. Executed tests are stored into a database. Known results may be compared to test results. Regression testing is supported. From the tester's point of view this is probably the most important service of RITA. However, from our point of view it is not that interesting.

4.6 Statistics

Finally, RITA will generate various reports and graphical presentations of the test process and tested software. The first prototype will include

at least various coverage percentage based on executed tests, graphical coverage information of chosen classes, and information about template and hook objects, and their test states.

5 Conclusion and Future Work

In this paper we have presented the prototype functionality of RITA, a framework integration and testing application. We strongly believe that RITA functionality is useful and even necessary for framework-based application testing. This is especially true in product line environments where lots of closely related programs are being developed and tested.

We are currently implementing the first prototype of RITA. The prototype will include the core functionality of RITA including a user interface, a profiler, a path identifier for hot spot coverage, some coverage analysis, and test execution. In the future versions we will implement more of the RITA functionality and also expand the core when necessary.

Bibliography

[1] M. Ardis, N. Daley, D. Hoffman, H. Siy, and D. Weiss. *Software product lines: a case study.* Software—Practice and Experience, 2000. Vol. 30, no. 7, pp. 825–847.

[2] T. R. Arnold and W. A. Fuson. *Testing "in a perfect world".* Communications of the ACM, 1994. Vol. 37, no. 9, pp. 78–86.

[3] R. V. Binder. *Design for testability in object-oriented systems.* Communications of the ACM, 1994. Vol. 37, no. 9, pp. 87–101.

[4] J. Bosch. *Product-line architectures in industry: A case study.* Proceedings of the 21st International Conference on Software Engineering (ICSE'99). ACM, 1999. pp. 544–554.

[5] N. Daley, D. Hoffman, and P. Strooper. *A framework for table driven testing of Java classes.* Software—Practice and Experience, 2002. Vol. 32, no. 6, pp. 465–493.

[6] M. Fayad, D. Schmidt, and R. Johnson. *Building Application Frameworks.* Wiley and Sons, 1999.

[7] M. Jazayeri, A. Ran, and F. van der Linden. *Software Architecture for Product Families: Principles and Practice.* Addison-Wesley, 2000.

[8] T. McCabe. *A complexity measure.* IEEE Transactions on Software Engineering, 1976. Vol. 2, no. 4, pp. 308–320.

[9] J. D. McGregor and T. D. Korson. *Integrated object-oriented testing and development processes.* Communications of the ACM, 1994. Vol. 37, no. 9, pp. 59–77.

[10] J. D. McGregor and D. A. Sykes. *A Practical Guide to Testing Object-Oriented Software.* Addison-Wesley, 2001.

[11] W. Pree. *Design Patterns for Object-Oriented Software Development.* Addison-Wesley, 1995.

[12] *A Framework for Software Product Line Practice—Version 3.0.* Software Engineering Institute, Northorp, Linda M. (director), Carnegie Mellon University, Pittsburgh, PA, USA, 2001. `http://www.sei.emu.edu/plp/framework.html` [March 16, 2002].

[13] Y. Wang, G. King, M. Fayad, D. Patel, I. Court, G. Staples, and M. Ross. *On built-in test reuse in object-oriented framework design.* ACM Computing Surveys, 2000. Vol. 32, 1es, pp. 7–12.