RITA Environment for Testing Framework-based Software Product Lines

Raine Kauppinen and Juha Taina

University of Helsinki
Department of Computer Science
P. O. Box 26 (Teollisuuskatu 23)
FIN-00014 UNIVERSITY OF HELSINKI
e-mail: {Raine.Kauppinen, Juha.Taina}@cs.helsinki.fi

Abstract. A software product line can be used to implement a software product family that is a set of software products sharing common features. A natural implementation strategy for a software product line is an object-oriented application framework. When software product lines are tested, tool support is essential. Also, there is a need for product line specific testing methodology. In this article, the RITA environment for testing framework-based software product lines is introduced.

1 Introduction

A *software product line* can be used to implement a *software product family* that is a set of software products that share common features. A natural implementation strategy for a software product line is an *object-oriented application framework* [5]. It is a partial design and implementation of an architecture, and basic functionality for an application that belongs to a given family. As such, frameworks can be used to capture commonalities between different applications in a software product family while allowing variation among its members. A framework-based application may be executable even if some of the application specific parts are missing, if the framework itself is executable and application specific parts that are not available can be replaced with stubs. When a framework is used to implement the core of a product line, *framelets* are often used as well. A framelet is a mini-framework that contains less than ten classes and has a simple, clearly defined interface [16].

The current testing methods for frameworks and product lines are quite immature, so there is a need for new testing methods. The testing process should be supported by testing tools and automated as much as possible. In this article, we introduce the design of the *RITA* environment that can be used in testing of framework-based product lines. This article is organized as follows. Section 2 is an overview to related work relevant to this article. Section 3 is a brief introduction to the current state-of-the-art of product line testing. Section 4 introduces the RITA environment. Section 5 discusses the current status of RITA and future work. Section 6 concludes this article.

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 [6]. Also, SEI has developed a framework for software product line practice [15]. In addition, some case studies involving software product lines have been made [1,2]. The use of application frameworks to implement product lines has been studied throughly by several research projects [3,5,18].

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 [3,12,19], but there is also growing demand for a well-defined product line testing process and methodology including tool support [17]. One approach to product line testing is presented in the SEI's framework for software product line practice [15]. Also McGregor and Sykes have defined a testing process and introduced methodology and tools that can be used in the product line approach [11,13].

3 Testing in the Product Line Approach

Product line testing, like the traditional testing of a single application, is usually performed according to the standard V-model [4,11]. However, if the V-model is to be used, product line testing process has to be integrated with a higher level product line business process [9].

Unlike the V-model that has become a standard approach for testing a single application, there is no standard process framework that would integrate the overall product line business process to the product line testing process. Instead, there are several process frameworks that can be used. For example, the ESAPS and CAFÉ projects [10] and SEI have their own process frameworks [15].

Integrating testing to product line process frameworks has proved to be somewhat problematic in practice [8]. It seems to be unclear from the business point of view where testing belongs in the overall product line process. For example, both CAFÉ's and SEI's process frameworks imply that testing is performed according to the standard V-model, but neither framework explicitly integrates the V-model to the overall product line process. However, SEI's process framework has a detailed description of different testing phases related to development phases of the process framework.

According to the existing product line process frameworks, testing is based on generating, managing and using *reusable test assets* that contain test suites and other *test artifacts* [11]. To effectively manage test assets, a *test asset repository* is needed. Reusable assets contain artifacts that can be reused throughout the lifecycle of a product line. The assets include, for example, documents, use cases, scenarios, classes and other software components [10]. Testing related assets include, for example, test plans, test suites and test reports [13].

The asset repository can be separated or integrated. The standard way to implement the asset repository is to separate and manage application code and

test assets with a database management system (DBMS). Next to this, object-oriented application frameworks also provide a simple way to integrate the application code and the assets via inheritance mechanism. This idea has been applied in the form of *built-in tests* [19]. The idea is to integrate tests into classes of an application framework. In this way, tests can be inherited to the classes that implement the application framework. In other words, built-in tests can be used as default properties or services of the classes in an application framework [7].

While test assets and their use in the product line approach have been studied, the current product line practice does not describe which testing methods should be used when product lines are tested [8]. The asset repository and test artifacts are used to support testing, but no product line specific testing methods have been presented apart from the build-in tests. Instead, it is assumed that the methods are similar to the traditional object-oriented testing methods. Also, product line testing is lacking tool support. Although traditional testing tools designed for testing of a single application can be used, there is a need for product line specific testing tools.

4 RITA: fRamework Integration and Testing Application for Product Lines

The *RITA environment* for product line testing is designed to tackle the key question of how the traditional object-oriented testing methods should be used when product lines based on object-oriented application frameworks are tested. RITA provides an environment specifically designed for testing of framework and framelet-based product lines. It includes services for interface class identification, code profiling, coverage criteria analysis, driver and stub generation, test management, and statistics. The environment can be used, on one hand, to apply the existing object-oriented testing methods in the product line context, and, on the other hand, to explore new, product line specific testing methods.

4.1 RITA Inputs and Outputs

The main input for RITA is the application code consisting of the code from a framework, framelets and application specific code elements. In addition, the environment needs information about the interfaces of between the framework, framelets and application specific code elements. This information describes interfaces that are used to extend the framework with framelets or application specific code. A framelet can also be extented with application specific code via similar interface. An interface used to extend a framework or a framelet consists of *templates* and *hooks* [7,16].

A template is a class that contains a *template method* that has references to one or more abstract *hook methods* residing in one or more abstract hook classes. Application specific parts or framelets are plugged into a framework or a framelet by instantiating a hook method. A hook method can be instantiated by implementing the abstract hook class it resides in or by overriding the abstract

hook method. The latter option can be used only if the template method and the hook method reside in the same class, that is, when the template class and the hook class are the same class. A hook is *connected* if one or more of its hook methods are instantiated and a template is *connected* if the hook methods referenced from it are connected.

Testing related inputs of RITA are test materials and drivers and stubs needed for testing of the possibly partial application. These can be either generated manually or by the RITA environment. Outputs of the environment include test results that contain information about passed and failed tests, statistics of the tests (for example, different coverages), new test cases generated by the environment (based, for example, on template and hook interfaces) and new drivers and stubs that are generated by the environment if needed [17].

4.2 RITA Views

The RITA environment provides four views to the application under testing. The views are *framework view*, *template view*, *class view* and *method view*. Each view offers view-specific testing services for the application. Figure 1 shows an example application to be tested with RITA. The application is a real-time database management system application derived from a framework-based DBMS product line. The framework of the example provides database services and offers three hooks for application specific expansions. Two of the hooks are extended, both with framelets. Framelets provide real-time capabilities and disk management functions. The framelet providing real-time capabities has three hooks of which two are extended with application specific code. One extension provides services for scheduling and the other provides service for transactions. This example is used in the following chapters to describe the four views of the RITA environment.

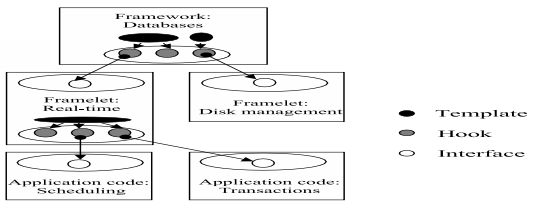

Fig. 1. An example of a real-time DBMS application of a framework-based DBMS product line

The framework view is the product line level view to the application under testing. It offers services for testing between framework elements (the framework, and framelets and application code elements connected to the framework). The framework view includes, for example, application-level black-box testing services and statistics of application-level coverages. At this view frameworks, framelets and application code elements and their connections are shown. Unconnected hooks are also visible, but the class hierarchy of elements is not shown.

The framework view of the application in Fig. 1 is shown in Fig. 2. The view shows the framework, the two framelets and the two application code elements of the application. When any of these elements is clicked, corresponding template view is opened. The view also shows the two unconnected hooks of the application.

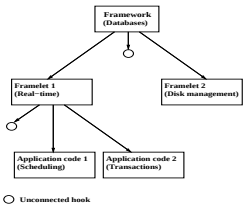

Fig. 2. The framework view of the example application shown in Fig. 1

The template view shows a UML class diagram of a framework, a framelet or an application code element. The diagram includes template and hook classes of the element, so interfaces between the element in question and other framework elements is visible at this level. The template view offers services for interface testing. The services include automatic test case generation for interface testing between framework elements, and automatic driver and stub generation for framework element interfaces among other things.

The template view of the framelet providing real-time capabilities, `Framelet 1` in Fig. 2, is illustrated in Fig. 3. The view shows the UML class diagram of the framelet. In the framework area, the abstract hook class `FrameworkHook` is shown. Its counterpart, the instantiated hook class `FrameletHookInstanciation1` that is used to extend the framework is shown in the framelet area of the view. If the framework area surrounding the class `FrameworkHook` is clicked, the template view of the framework is shown. If the class itself is clicked, the corresponding class view is shown.

In addition to the instantiated hook class, the framelet area shows the other classes of the framelet. They include classes that implement the functionality of the framework (`FrameletClass1`, `FrameletClass2` and `SimpleStateMachine`), and also template and hook classes that provide the three hooks of the framelet. First hook is provided by classes `FrameletTemplate1` and `FrameletHook1`, second by the class `FrameletTemplateHook1` and third by classes `FrameletTemplate2` and `FrameletHook2`. The hook that uses the classes `FrameletTemplate1` and `FramletHook1` is not implemented. When a class in the framelet area is clicked, the corresponding class view is shown.

The class view offers two presentations for a class. The first shows the method references of each method of the class and the other shows the code of the class. The class view offers services for class testing. It includes standard object-oriented testing services. The services include, for example, interface testing of a class, testing of a class state and method collaboration testing. Driver and stub generation is supported by providing places where manually generated drivers and stubs can be plugged in. The exact services of this view are under design and will not be implemented in the first prototype of RITA.

The class view of the class `SimpleStateMachine` in Fig. 3 is shown in Fig. 4. Figure 4 shows both presentations of the class view: the method reference presentation of the class is shown on the left and the code presentation on the right. In this case, the class contains only three methods. The constructor of the class, method `SimpleStateMachine` has no references to other methods. Method `changeState`, however, uses the method `isValid`. The methods of this class do not contain method calls to other classes. Such references are shown in the method reference presentation so that the arrow from the calling method is connected to the class which holds the called method. When a method is clicked in the method reference presentation or in the code presentation, the corresponding method view is shown.

The method view offers services for unit testing at the method level. This view includes various coverage criteria (for example, code-based, state-based and constraint-based coverages) and a list of recognized independent paths through a method. Also this view provides places for manually generated driver and stubs.

The method view offers two presentations for a method. The first is a flowchart view to the method and the other is the code of the method which could be used, for example, in illustrating the statement coverage in addition to showing the actual method code. The method view of the method `changeState` in Fig. 4 is shown in Fig. 5. The figure shows the flowchart of the method on the left and the code on the right.

The four views of RITA provide a hierarchical way to use test cases. For example, the results of a test case used at template view to test hook and template interfaces can be seen and evaluated in the lower levels as well. This is true for test cases used at every view of RITA. This hierarchical approach provides additional information about testing, because every view has specific testing features. At class view, for example, it is possible to see the method references invoked by a test case used originally at template view. Furher information about the cover-

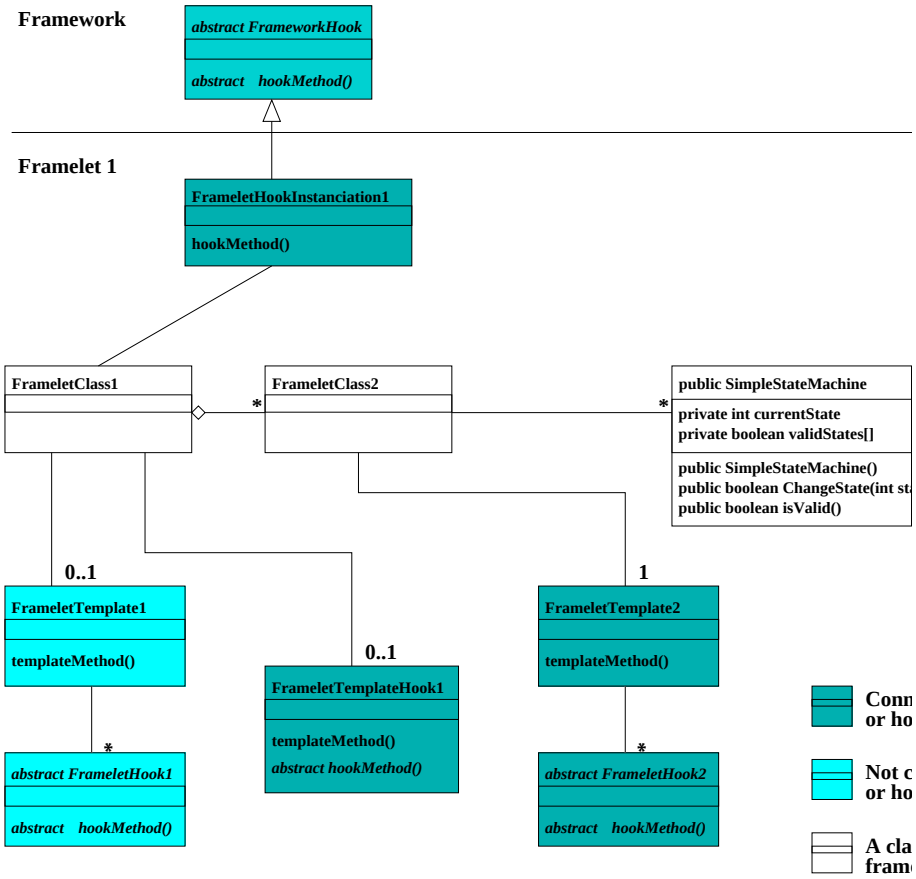

Fig. 3. The template view of `Framelet 1` in Fig. 2

age of the test case can be obtained from the method view, where, for example, the statements of any method the test case has executed are shown.

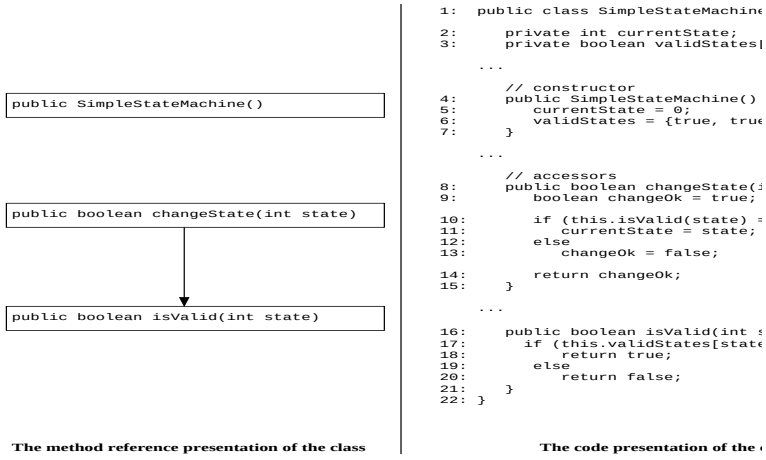

Fig. 4. The class view of `SimpleStateMachine` in Fig. 3

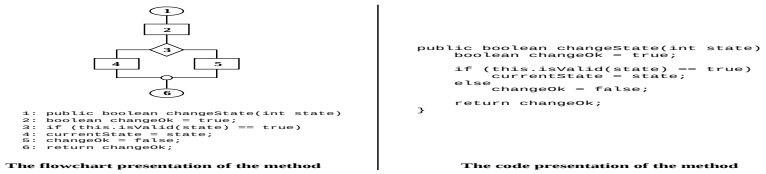

Fig. 5. The method view of the method `changeState` in Fig. 4

4.3 Product Line Testing Process Supported by RITA

The testing process supported by RITA is illustrated in Fig. 6. The process requires a test plan based on requirements as input, since RITA does not support test planning or requirement specification. Test cases are generated mostly manually, but the environment can generate additional test cases automatically based on, for example, hook and template information. After test cases are generated

and selected for execution, the environment scripts and executes the tests. After tests are run, results can be evaluated based on the test report generated by the environment. RITA also manages testing assets thoughout the process, for example, by maintaining a test asset repository.

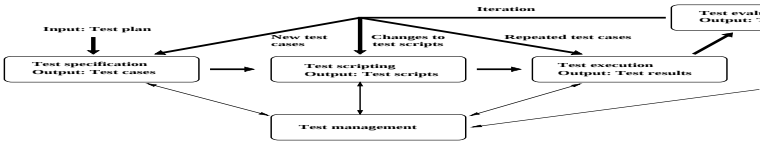

Fig. 6. Testing process supported by RITA

The testing process is iterative. Based on the results of executed tests, new test cases can be generated and tests scripts may be updated. The existing tests can also be repeated without change, for example, when regression testing is performed. When new test cases are generated, either manually or automatically, the process is repeated for them. It is also possible that test scrips need to be changed, for example, because interfaces of the application under testing have been changed.

The RITA environment can be used to test a single application. In this case, the test plan is designed specifically for the application and the testing process is iterated according to its lifecycle. However, the RITA environment is focused on testing of product lines instead of testing of single applications. When product lines are tested, the test plan covers the entire product line. It specifies how the framework of the product line is tested before applications are derived from it. The test plan specifies also how the derived applications are tested. In this case, the testing process is iterated throughout the lifecycle of the product line. In this way, actual product line testing can be performed and existing test assets can be effectively reused.

5 Current Status and Future Work

We have recently finished the first prototype of the RITA environment. The prototype implements framework, template, class and method views and provides

code profiling services, partial coverage criteria analysis (hook and template coverages are not implemented) and basic test management functionality and statistics. At this point it seems that there is a clear need for both the environment itself and for new testing theory aimed at binding the high-level product line testing process frameworks to the low-lewel testing methodology in the product line context.

One of the main advantages of RITA is that it visualizes the structure of a framework-based product line application. The environment also visualizes the parts of the application that are covered by executed test suites. The visualization helps to identify the problem areas of framework-based applications. The framework view helps to understand the structure of the application and the relationships between a framework, framelets and application specific code elements. The template view illustrates the interfaces between different parts of the application. The class and method views provide more traditional aids to testing. For example, the class view can be used to find dead code segments and unnecessary references to other classes or methods. The method view can be used to gain detailed information about the algorithms used in the application.

In addition to visualizing framework-based applications, RITA can be used as a reverse-engineering tool for applications that are not framework-based, since the environment is able to generate framework, template, class and method views for these kind of applications as well. In this case, the framework view shows the entire application as black box, and template view shows the UML class diagram of the entire application. The class and the method views are similar to those generated from framework-based applications.

With RITA, we have applied traditional object-oriented measures of coverage at the component level of product line testing. In addition, product line specific measures of coverage that can be used to evaluate the overall testing coverage over a product line have proven to be useful. Therefore, we propose two new coverage criteria for framework-based product line applications: *hook coverage* and *template coverage* [8]. Template and hook coverages define how much of the functionality of hand-written application code elements and framelets extending an application framework via hooks have been covered with existing test suites. Informally, hook coverage is the statement coverage of all the connected hooks provided by executed test suites. Template coverage is the number of references from all the connected template methods to connected hook methods that are covered by executed test suites. Template and hook coverages can be used to measure the coverage of an entire product line application or a single framework or a framelet. They can be used to measure the progress of the implementation in addition to measuring the adequcy of the testing performed.

Currently, information about templates and hooks of the application under testing has to be generated manually. External tools can be used to import this information to RITA. For example, if the framework of the application is generated with a framework editor, the hook and template class information may be imported from the editor. One such editor is the *FRED* tool [18]. Another pos-

sibility is to import the information from a tool that can regognize the template and hook classes of the framework as patterns, such as the *MAISA* tool [14].

The template and hook information makes it possible to, on one hand, to test thoroughly the core of a product line (the framework) as well as entire applications derived from it. On the other hand, it is possible to partially test a product line application. Application specific parts can be tested separately starting from interface where they will be plugged in. This requires driver support that RITA also provides. Similar approach can be used to test framelets. A driver replacing the template is used to apply the instantiated hook of a framelet. In this, stubs may be required as well, if any of the framelet extensions needed are not available.

A key area of our future work with RITA is to identify templates and hooks automatically from the source code. We will implement a template and hook class identifier that first recognizes potential templates and hooks. The identifier then shows a list of candidates and the end user chooses the actual templates and hooks.

Other areas of future work include support for product line specific coverage criteria such as hook and template coverage, implementation of automatic driver and stub generator and advanced test management services with a test asset repository. We expect that the RITA environment will evolve into a useful environment for product line testing that can be used also in practical software engineering projects in addition to its research use. This will take time and resources and require co-operation with industrial partners. Also, thorough evaluation of the RITA environment and the theory behind it is needed.

6 Conclusions

The state-of-the-art of product line testing is immature, so there is a clear need for more mature testing methodology. Most of the research so far has concentrated on the product line testing process and on the assets that can be reused thoughout the process. Regarding these issues, product line process frameworks and ideas of asset repositories have been formulated. However, there is a gap between the product line testing process and the practical testing methods. It is not clear which object-oriented testing methods can be effectively used in this particular context. Testing also lacks necessary tool support and automation that are essential to the product line approach.

However, the work to tackle the problems encountered is under way. The product line testing process is under extensive research and new testing methodology is being developed. The CAFÉ project and its followup projects in Europe and SEI in the USA are currently studying new testing theory and deriving practical methodology and tools to be used with product lines. Examples of the work done in the CAFÉ project are the development of the RITA environment for testing of framework-based product lines and the definition of hook and template coverages.

Acknowledgements

This work was funded by Nokia Research Center as a part of the ITEA project CAFÉ (project number 00004). The authors would also like to thank the other members of the RITA project, professor Jukka Paakki and research assistant Antti Tevanlinna from the University of Helsinki, for their valuable comments and ideas regarding this work.

References

1. Bosch, J., Product Line Architectures in Industry: A Case Study. *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, Los Angeles, California, USA, May 1999, 544-554.
2. Cohen, S., *Product Line State of the Practice Report*. Technical Note CMU/SEI-2002-TN-017, Carnegie Mellon University, Software Engineering Institute, October 2002.
3. Fayad, M., Hamu, D., Brugali, D., Enterprise Frameworks Characteristics, Criteria, and Challenges. *Communications of the ACM*, Volume 43, Number 10, October 2000, 39–46.
4. Fewster, M., Graham, D., *Software Test Automation – Effective Use of Test Execution Tools*. Addison-Wesley, 1999.
5. van Gurp, J., Bosch, J., Design, Implementation and Evolution of Object-Oriented Frameworks: Concepts and Guidelines. *Software – Practice and Experience*, Volume 31, Number 3, March 2001, 277–300.
6. Jazayeri, M., Ran, A., Linden, F. (eds.), *Software Architectures for Product Families: Principles and Practice*. Addison-Wesley, 2000.
7. Jeon, T., Seung, H., Lee, S., Embedding Built-in Tests in Hot Spots of an Object-Oriented Framework. *ACM SIGPLAN Notices*, Volume 37, Number 8, August 2002, 25–34.
8. Kauppinen, R., *Testing Framework-based Software Product Lines*. Master's Thesis, University of Helsinki, Department of Computer Science, 2003, to appear.
9. van der Linden, F., Software Product Families in Europe: The Esaps & Café Projects. *IEEE Software*, Volume 19, Number 4, July/August 2002, 41–49.
10. van der Linden, F., ESAPS-CAFÉ Inputs. *Proceedings of the 3rd ITEA Symposium*, Amsterdam, Netherlands, October 2002, URL: http://www.itea-office.org/symposium/ [March 21, 2003].
11. McGregor, J., *Testing a Software Product Line*. Technical Report CMU/SEI-2001-TR-022, Carnegie Mellon University, Software Engineering Institute, December 2001.
12. McGregor, J., Korson, T., Integrated Object-Oriented Testing and Development Process. *Communications of the ACM*, Volume 37, Number 9, September 1994, 59–77.
13. McGregor, J., Sykes, D., *A Practical Guide to Testing Object-Oriented Software*. Addison-Wesley, 2001.
14. Nenonen, L., et al., Measuring Object-Oriented Software Architectures from UML Diagrams. *Proceedings of 4th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, Sophia Antipolis, France, June 2000, 87-100

15. Northorp, L. (director), *A Framework for Software Product Line Practice – Version 3.0*. Software Engineering Institute, Carnegie Mellon University, 2001, URL: http://www.sei.emu.edu/plp/framework.html [March 21, 2003].
16. Pree, W., *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, 1995.
17. Taina, J., Paakki, J., Kauppinen, R., RITA - a fRamework Integration and Testing Application. *Proceedings of the Finnish Data Processing Week (FDPW'02)*, Petrozavodsk, Russia, July 2002, to appear.
18. Viljamaa, A., *Pattern-Based Framework Annotation and Adaptation – A Systematic Approach*. Licentiate Thesis, Report C-2001-52, University of Helsinki, Department of Computer Science, 2001.
19. Wang, Y., et al., On Built-in Test Reuse in Object-Oriented Framework Design. *ACM Computing Surveys*, Volume 32, Number 1 (electronic supplement), March 2000, 7–12.