Product Family Testing with RITA

Antti Tevanlinna
University of Helsinki
Department of Computer Science
P. O. Box 68 (Gustaf Hällströminkatu 2 B)
FIN-00014 UNIVERSITY OF HELSINKI

Abstract

Product family engineering is a developing discipline where testing remains a central task. Product family testing, however, has received attention only recently despite the problems caused by the restructuring of the overall development process to a product family engineering mode. We discuss the current state of product family testing and introduce the framework integration testing tool RITA (fRamework Integration and Testing Application). We present the architecture of RITA, which allows flexible inclusion of new testing services and the evolution of our tool. We present the current state of RITA and describe how it can be leveraged to measure, test, and analyze the testing executed on framework- and component-based product families.

1 Introduction

A *software product family* is a set of software products that share common features. The commonality across members of a family is useful in the design and implementation of products. It is possible to develop products as a family of applications instead of considering the applications as separate. Since the entire product family is developed as a whole, the commonality of the products can be managed.

The main advantage of product families is the reduced time to market and development costs once the product family for the application domain is functional. Another important advantage is the large-scale reuse of software when the commonalities in the given application domain are captured in the product family. In this sense, the *product family approach* in software development is a natural step from hand-made software products tailored for each customer to mass-production of software products [Nor03].

Product family engineering is usually divided into domain and application engineering tasks [Nor03, vdL02]. The role of domain engineering is to produce common software in the applications. Produced by domain engineering, a natural core of a product family is a set of reused software assets, such as application frameworks [FHB00, vGB01] and software components [Szy98, Bos00], in which companies have captured commonalities of applications in the domain where they operate. The role of application engineering is to be a consumer of domain engineering producing the final applications based on the common software of the product family.

The product family approach has a profound effect on the overall development process, which in turn calls for changes in the testing processes as parts of the overall process. Because product family testing is a new research area, it is understandable that many of its aspects have not yet been investigated. The managerial and business views of product families have been under extensive research [Kau03]. Testing related results include the product family process frameworks created by the SEI [Nor03] and the ESAPS and CAFÉ projects [vdL02], and the ideas of reusable test assets and their management [McG01].

In our previous work [TTK04], we recognized a gap between the overall product family engineering process and the implementation of testing as a part of cost-efficient quality assurance for product families. The gap is mainly due to problems in dividing testing tasks in a product family testing process and the increase in the importance of test management that is required to handle the extra complexity of the product family approach. The simplest solution is to rely on testing in application engineering, which is expensive in the family scale because of the redundant testing of the common assets.

We promote extensive testing of the reusable assets in domain engineering and product testing based on the domain testing. In domain engineering, configurations of the common assets expected to be formed should be tested well. The domain effort minimizes the testing of the common assets needed in application engineering. Product testing should be concerned with product-specific testing objectives plus the integration of the common assets and product-specific software.

Before removing the seemingly duplicate testing from the application engineering task, the quality of the reused assets must be assured. Evaluation and measurement of quality-related properties including the extent of the testing performed during the lifetime of the assets are therefore required in an incremental product family testing process.

In this paper, we present the current state of the RITA (fRamework Integration and Testing Application) testing tool. We discuss the distinctive properties and problems of product family and framework testing, and the requirements on tools in this domain. By building RITA, we have investigated the possibilities of tool support for integration testing of framework-based applications and product families. We present the architecture, the current state, and the future plans of RITA.

We have created a small product family for stream processing to illustrate how frameworks can be used in the realization of product families. The two products of the family are simplified grep- and tr-like utilities (hereafter grep- and tr-application) that share control logic and adapt the required stream processing. We use the family as a running example throughout the paper.

The rest of this paper is organized as follows. Section 2 discusses product family and framework testing. Section 3 discusses the requirements on testing tools in this domain. In Sections 4 and 5, our approach and the current status of our tool are presented. Finally, Section 6 presents future improvement directions for RITA, and summarizes and concludes the paper.

2 Product family and framework testing

The division in domain and application engineering of product families is captured in *domain asset development* and *product development* activities. Product development is a consumer of *reusable assets* produced in the domain asset development. Products are made leveraging the reusable assets. Feedback from product development to core asset development is essential as well as the overall management of product family engineering (Figure 1).

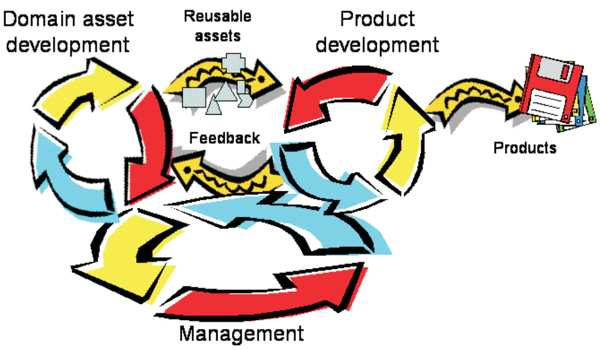

Figure 1: Essential product family engineering activities [Nor02].

Enabling variability, components may be developed as called frameworks, also known as framelets, that usually contain less than ten classes. In the approach, the components have a common structure and logic (cold spots) and some part of structure and logic is specified later (hot spots). When needed, the component can be adapted to function as required by extending abstract classes or by implementing interfaces that function as hooks [Pre94] in the framework. Frameworks often facilitate this by providing a collection of ready-made extensions to their hot spots which shifts the nature of framework adaptation from extension towards composition of objects. When the framework invokes services through methods defined in the hot spots, virtual binding delivers the invocation to the adapting code based on the runtime class of the object whose services are invoked. One of the best know collections of this kind of components is the Swing toolkit in Java where almost every user interface component can be adapted by changing the model, the controller, or the view.

The framework of our example product family contains a control loop that processes a sequence of tokens. It iterates the tokens to which an abstract input has been cut. The io-functionality in the family has been wrapped to reused components. The input component reads the standard input stream and files, and it produces line tokens. The output component prints to the standard output stream. Tokens are matched against product-specific criteria and the found matches are replaced by the adapting application code (Figure 2). The adaptations decide also on decorations added to tokens and matches and whether to output the tokens. The grep-application seeks and outputs lines containing the letter `a`. The tr-application is more sophisticated: it replaces instances of the string `a` with the string `->b<-` and adds line numbers to the output produced.

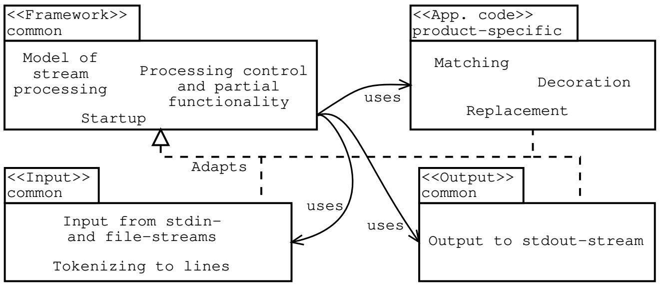

Figure 2: Division of functionality in the example product family.

The domain assets are tested and here we have the *domain tester's problem*. A domain tester tries to minimize the application testing. For the tester, it is difficult to ensure that the software works correctly in possible usage scenarios, all of which are not yet specified. The domain tester will find complete integration and system testing according to the V-model unfeasible because the implementation of variations in the family is not yet specified. In a component-based approach, trying to test the combinations of components leads to exponential growth of tested configurations [McG01, MvdH03].

In application engineering, the products created are tested and the *application tester's problem* becomes apparent. Based on the domain testing and testing of other applications in the family, an application tester tries to figure out what testing would be adequate for the particular product under test with minimal cost. To decide on how much we can rely on the past testing of the common assets and what aspects of the product must be tested in the application testing, we have to perform rigorous test planning and thorough analysis of adequacy of testing. Let us suppose that the second product of the example family is being implemented. The framework has already been tested in the domain engineering and in the first product using the framework. Blind trust on the correctness of reused assets can however lead to failures in products and sometimes serious consequences [Wey98].

In our earlier work, we identified four strategies to model product family testing [TTK04]: *testing product by product*, *incremental testing of product families*, *reusable asset instantiation* and *division of responsibilities*. We also noted that the strategies must be mixed and matched to test the right aspects of the right assets cost-effectively. A case study in Nokia Mobile Phones contains examples of the strategies [Jaa02].

The example shown in Figure 3 summarizes the considerations in product family testing. In the example organizational structure, we have a testing team both in every application engineering unit and in the domain engineering unit. The testing responsibilities are divided between the teams. Regression testing techniques are applied to incrementally test the evolving functionality, and domain testing produces half-ready reusable test assets to support product testing. In this case, the product testing teams need to be aware of what is tested in the other products and what is tested in domain engineering. The domain testing team needs to be aware of what is tested in the product testing teams to produce usable test assets. Much information will have to be exchanged to achieve this. Lack of coordination may also be a problem. There is by default no all-knowing party to tell what has been tested and what still needs to be tested. Synchronization and management of the processes is therefore essential.

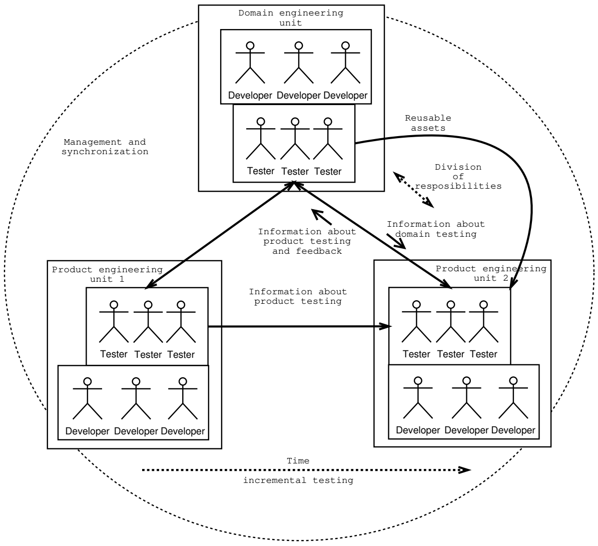

Figure 3: An example of a product family organization, testing related asset and information flows in it (solid arrows), and process-related considerations (dotted lines).

Component-based integration testing is inherent to product family integration testing. An organization where a team or a software factory is responsible for each component and its internal testing will eventually have to perform component integration. When the components, like the example framework and its adaptations, are brought together for integration, testing concentrates on with their cooperation and interfaces, because the components are already tested as units.

Many criteria have been defined to test polymorphism in object-oriented languages [CK99, AO00, Bin00]. One of the main integration testing concerns in testing a framework is that both the framework and its extensions make correct assumptions of each other. In other words, the framework should not use any of its extensions incorrectly and none of the extensions should adapt or use the framework incorrectly. At a minimum, structural integration testing calls for exercising all the bindings from a template method to a hook implementation method. Other aspects of testing can be included by expanding the criteria towards data-flow and state-based testing.

Because frameworks are a means to implement a product family, the product family testers' problems apply. A framework is difficult to test, because applications using it may not be available. Without extensions to the hot spots, the framework must be tested using stubs instead of actual adaptations. System and acceptance testing are also impossible without an example or a reference product. On the other hand, an application tester will find it difficult to evaluate the thoroughness of testing performed on the framework even if related test reports and test suites are available. In our example, testing the grep-application would require much attention on the correctness of the framework while the tester of the tr-application could concentrate more on the adaptations. To trouble the tester of the tr-application, functionality not used and tested in the grep-application still exists in the replacement functionality of the framework.

3 Requirements on framework-based product family integration testing tools

Automated testing and the use of testing tools is essential in product family and framework testing. This decreases the effort when reusing test assets and makes the complex testing processes more manageable. The extra cost of automating tests is more worthwhile because the common tests are extensively reused and re-executed.

There are numerous mature testing tools available nowadays, but, despite the abundance, product family and framework testing lack efficient tool support. The problem with testing tools is that few of them address the problems of product family or framework testing. For example, there are separate tools that provide entire test execution environments such as the *JUnit framework* [GB99], the *STA Framework* [Ran02], and the *TestDirector* [Mer03]. Together, these cover test execution on all levels of the V-model. Usually, tool support for anything other than test execution and test case template generation (such as in the Eclipse IDE [Ecl04]) is random.

In the product family context, we need product family specific testing tools. The testing tools should support performing the tasks involved with a cycle in testing process and help to effectively manage the created test assets to be reused. Tool support should be extended from the execution of tests and the analysis of their results to ultimately integrate the whole process of product family testing [KT03].

Scale is the first obstacle for a tool for managing the overall testing of a product family. Testing of a product family is a large-scale effort and the visible level of detail should be selected accordingly. Plain information about classes or test cases is not informative on the product family scale but the tools should present a higher level of abstraction.

Integration testing techniques are based on creating partial configurations of a system and testing them until the complete system is assembled. Thus, automating configuration definition and assembly would be a great help in integration testing. The integration effort is reduced when a tester is additionally supported in selecting the relevant test cases and in executing them on the configuration under test.

The common software should be presented distinctly from the product-specific software. Their relation should also be presented. This implies that a tester needs to identify the framework of the family, its templates and hooks, hook implementations, and other application code. The framework, the templates, and the hooks belong to the common software while the rest are of product-specific software. Once the two sides of the application are tested individually, the tester should concentrate on the interface and the mechanisms under test that relates the two together via hot spots. Comprehending the dynamics of virtual binding is difficult, and, on the framework scale, a tool that helps in managing the hot spots, their extensions, and the testing of the both would be valuable.

An engineer should be able to work with the product family as well as with separate products. Sometimes one specific product gains most attention, but, especially in domain engineering, we need to have a view of the whole product family. A domain tester evaluating the reliability of the common assets is more interested in the accumulated test coverage during the whole lifespan of the assets than of test coverages of a single product.

Working with product families, a common element should be treated as a single entity although it may be copied to every product. Conceptually, many properties of a set of products can be combined creating a single view from multiple similar views. For example, a diagram that contains structure of a product family with only one instance of each common component is more valuable for the domain engineer than a set of individual product-specific diagrams.

Working with multiple products, various set operations based on the products of the family would be valuable. Sometimes an engineer wants to work with all the assets of two products, sometimes she wants to work with the common assets of the products, and sometimes she finds it helpful to exclude assets belonging to some products.

Practitioners often require custom tool support because they need to use diverse approaches to product family engineering and to work in heterogeneous implementation environments. Product family engineering is still quite an instable domain where tools open for customization are needed before good all-in-one tools are in the market.

4 The RITA architecture and a model for product families

The RITA architecture is influenced by our need for both openness and integration. We wanted to cover as many tasks of product family and framework testing as possible. At the same time, RITA was envisioned to be an integrated all-in-one suite offering a set of services through a common access point. The ECMA model for tool integration [ECM93] (also known as the Toaster model) was a suitable reference architecture. The Software Test Environment Pyramid (STEP) reference model for testing tools [ER96] provided us guidance on dividing the functionality of a testing environment into service types included in our architecture.

Based on the Toaster and the STEP models, we designed an architecture that contains a model of a product family under test, service types to plug in testing functionality, and a user interface (Figure 4) [Tev03]. The service types provide test execution environment, static measurement, coverage analysis, graphic and textual visualization, repository management, and model management based on the structures defined in the model. On the top, the user interface layer unites the access points to services.

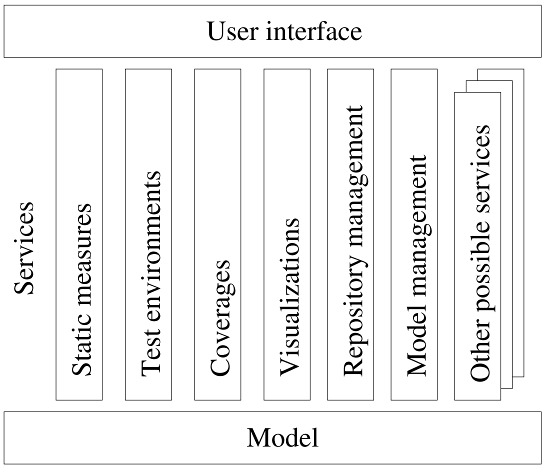

Figure 4: The RITA architecture.

The service types are abstract, which opens our architecture to different solutions within the abstract responsibilities of a service type. For example, we can select between a DBMS-based and a file-based persistence solution by changing the service configuration of RITA. No modifications or recompilation of RITA is needed. The service types also abstract away from language dependencies. To adapt RITA to be, say, a C++ test environment, one has to develop and include a new model management service, a new test environment, and the other possible services containing language dependencies. There is no need to modify RITA, because all that is needed is to provide a new service configuration. Adding new service types to RITA is also relatively simple provided the service implementations can operate on the model.

The architecture facilitates the evolution of the model by decoupling it from the user interface. The model can evolve without changes to the user interface, because the content is created by the services.

A language-independent model of a product family under test is the shared basis of the functionality of the services in RITA. The main elements needed to support the planned views at levels of abstraction envisioned in our earlier work [Kau03, KT03] are a product family, a component (or a framelet), a class, and a method (Figure 5). A method and a class are straightforward. A component is a set of classes. A product or a product family is a set of components. Other information includes compilation units, source code, test cases, and test results. We differentiated explicitly between common and the product-specific software by managing the commonality with the Composite design pattern [GHJV95] in the aggregate elements.

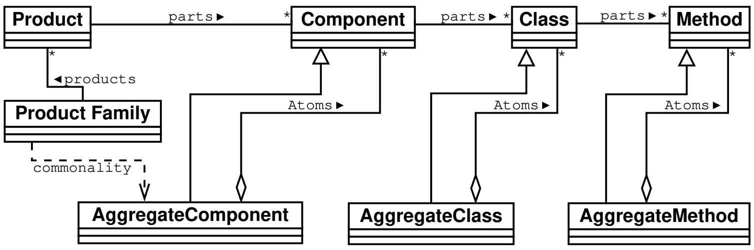

Figure 5: The main elements in the model of a product family under test.

Any relation between two software elements may be an association in our model. The relations between the elements are hierarchical entities built bottom-up. An association between two methods is any coupling, for example call-coupling or data-flow coupling, between the methods. We build upwards to class-level where two kinds of coupling exist. First, we have possible relations of two classes that include class extension and interface implementation. Second, we have method-level coupling from one class to another. When combined, an association between classes is a union of method-level associations from one class to another or a class relation. With the same grouping approach, an association between two components is a union of intra-component class associations (Figure 6).

The input of the model is governed by the services in our architecture. This together with the abstractness of the model allows representation of framework structures. We model the relations between the templates and hooks as special ”calls-hook” associations. We have similar relations for a method implementing a hook method (”overrides”) and the resulting virtually bound invocation from a template to a hook implementation (”calls-hook-implementation”). These relations are analyzed and created by the model management service when new products are imported to RITA.

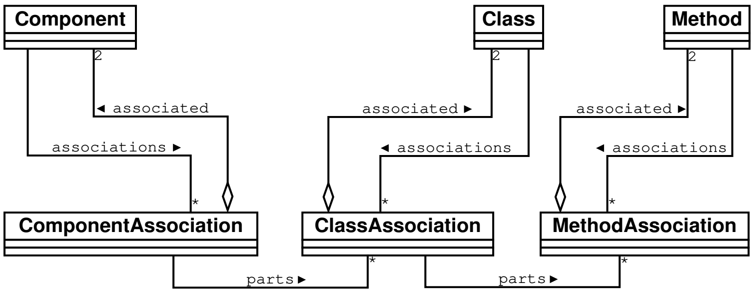

Figure 6: The associations between elements in the model.

The "uses"-association between the framework and input in Figure 2 is decomposable in RITA because an association between components is built from class-level coupling. A `InputSystem` class in the framework uses abstract stream input and tokenizing services adapted by the `Input`-component to read the standard and file io-streams in `StdReader` and `FileReader` classes, and to tokenize into lines in a `Linenizer` class. The class-level associations can be further decomposed into method call relations such as "`InputSystem.-getNextToken` calls-hook-implementation `Linenizer.next`".

5 Current state of RITA

The RITA user interface contains the structure of a set of tested products in a navigator on the left, and, on the right, RITA shows the details of the selection in the structure. The structure of selectable software elements consists of a product family, products, components, classes, methods, and associations between the elements. When the tester selects an element, RITA shows a collection of views and operations of the element. The tester can work with coverage and static measures, and various visual and textual presentations available for the selected element. The level of detail varies according to the abstraction level and each level presents a different scope in testing (Figure 7).

The levels of abstraction presented are similar as, in every view type, we present the internal structure and context of the selected software element, and provide testing operations. The presentations and operations provide assistance in test design, test case management and execution, and analysis of the testing executed.

The visual presentations and static measures in the views of RITA are useful in test design. We visualize a structural software element and other structures related to the element. Usually the both require testing. Thus, it is easy to explore the structures and to use the insights gained in test case design. We also provide measures of complexity and coupling that can be used in analysis of the structures during identification of software that needs special attention in testing.

After the planned test cases have been implemented, RITA can be used to

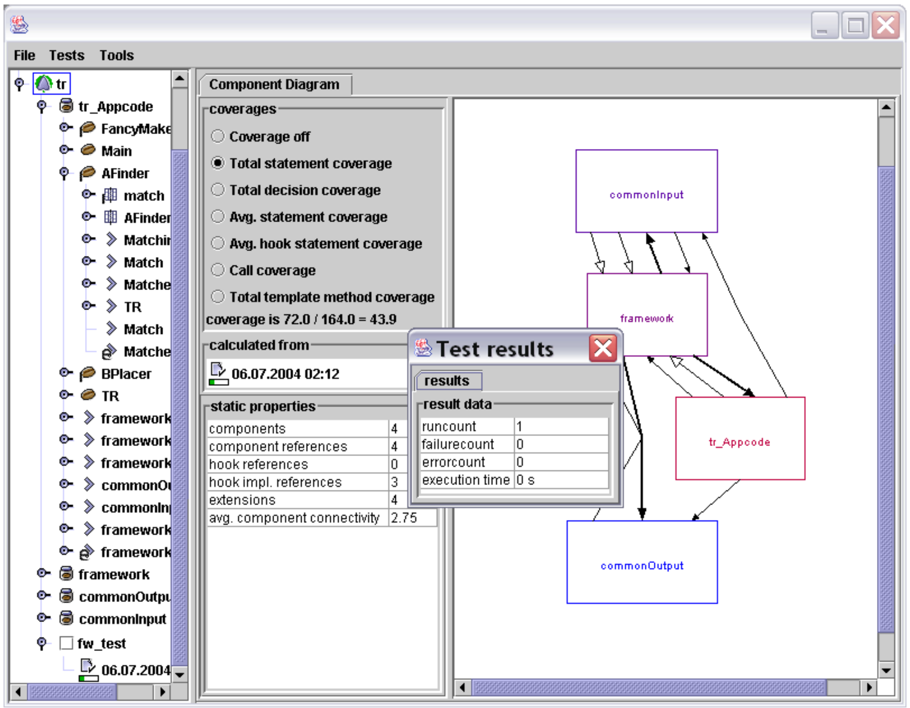

Figure 7: The RITA testing environment.

manage and execute test cases related to an element. The tester can associate test cases and test suites to elements in the navigator. This enables traceability between test assets and the structure of the family. When test execution is requested, RITA uses the capabilities of a test environment. By the test environment, the selected tests are executed on the products that contain the elements to which the tests are attached. The test results and execution traces are saved for future use.

RITA can be used to evaluate the extent of the testing performed using the various coverage measures. When the tester has executed a test suite, RITA can measure test coverage in the structure of the selected element. On all abstraction levels, RITA provides a set of coverage measures including statement, decision, and method invocation coverage criteria calculated from the structures visible. The measures can be used to evaluate the unit and integration testing of the elements. Coverage is visualized in the views by coloring covered structural elements blue and uncovered elements red. The partially covered elements are annotated proportionally with shades in between.

On the method level, the tester is presented with program code, flowcharts, and method graphs of the methods. This level serves mainly as a unit testing workbench for a method, but also method-level integration testing aspects are visible in the method graph that shows the method, methods referencing it, and methods called by it.

The class level shows a class with its context. We show the code of the class and its method graph. The graph contains the methods of the class and the related other methods. Thus, the usage of the class level views is similar to the method level.

The component level is concerned with the integration of a component. The tester is presented with two views of the selected component. A diagram in the first view shows only the internal class structure of the component. The second view contains the component and classes referencing or referenced by the component. The first view is useful in the internal integration of the component and the second view is helpful in evaluating the external integration of the component. The coverages on this level group the low-level coverage measures, like the statement coverage, based on the classes contained in the visualizations, and they highlight inadequately tested parts in the structure.

The levels of the product and the product family, where we show a set of components and their relations in a diagram (See Figure 7), are the highest level of abstraction in RITA. The components and their relations provide a complete view of the structure of a product or a product family and of the overall testing. On this level, RITA groups coverages based on the components, and therefore the views point out inadequately tested components. We have also implemented grouped coverage measures of the relations between components based on the hierarchical associations in our model. This is valuable in component-based integration testing, because testing of the integration of the components is explicitly presented and measured.

RITA contains special support for framework testing. As described in Section 4, the current implementation of our model contains the structural relations between templates, hooks, and hook implementations. The relations are visualized among other structural information in the views and in the navigator of RITA on the abstraction levels (Figure 8). To the tester, working with frameworks is no more than a special case of elements and relations between them. We annotate the templates, hooks and hook implementations in the navigator and emphasize the virtually bound relations in the graphic presentations by drawing them thicker. Based on the structures, we have also implemented services calculating the framework-specific Hook and Template coverage measures defined in our earlier work [Kau03] and static measures such as the number of hook methods in a class.

Already before any product-specific software is created, the tester of the tr-application can explore the structures and evaluate the testing of the framework and the grep-application. When the possibly still partial product is runnable, the tester executes a test of the framework that executes the adapted stream processor with three lines of sample data. Using the coverage calculation of RITA, the tester is presented with high-level presentations on the coverage achieved. We have captured the statement coverage achieved in the testing in Figure 7. As colors close to red indicate poor coverage, it is evident that testing is inadequate. Besides indicating the progress of the testing process, the coverage presentations are useful in improving the coverage. From the high-level presentations the tester

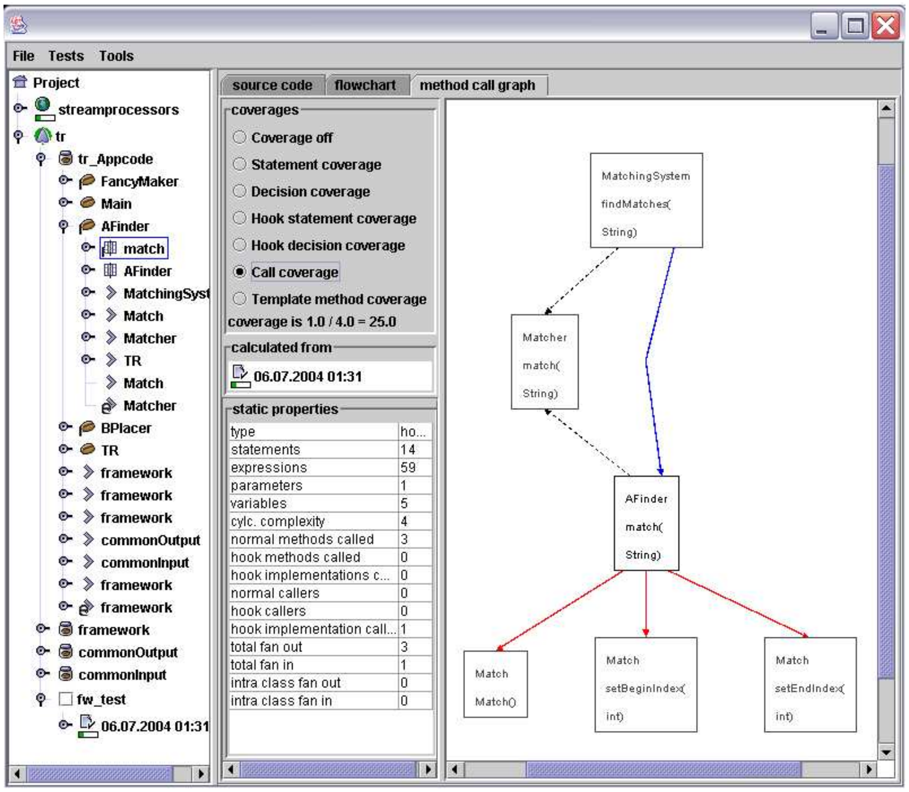

Figure 8: RITA visualizes the dynamically bound relations involved with hot spots in frameworks. The `MatchingSystem.findMatches` template method on the top invokes abstract services of the `Matcher.match` hook in the middle. Virtual binding, however, delivers invocations to the hook implementation in the `AFinder` class that supplies the actual implementation.

can progress to view and evaluate the coverage achieved in smaller scopes in the structure and pinpoint the actual places where more testing is required. In the tr-application, none of the test inputs match the criteria. This is also deductible from the test coverage of the adaptation of matching shown in Figure 8 as no `Match`-objects are instantiated.

Enabling working with multiple similarly structured products, product family aspects are present through the abstraction levels in RITA. We have ”product” and ”family” types of views and operations available for the selectable software elements. The family views and their operations are implemented using the aggregate elements in our model allowing us to define operations on a family instead of a product. Currently, we have implemented navigation, visualizations, static and coverage measures, and test execution on the family scale.

An operation or a measure on a family functions as either ”for every product” or ”for the combination (See Section 3) of the products”. For example, when a test is executed on a common method, it is actually executed on every product where the method exists. A combination containing the structure of the example family is captured in Figure 9. There, the three common components (`framework`, `commonInput`, and `commonOutput`) and their relations appear only once while the product-specific components (`grep_Appcode`, `tr_Appcode`) are also visible. The component diagram is annotated with test coverage achieved in testing of the applications using a common test case. The current coverage measure implementations in RITA calculate common elements as covered if they have been covered in any of the applications. We could also select to analyze the testing of the two products in the family view even when test cases are product-specific.

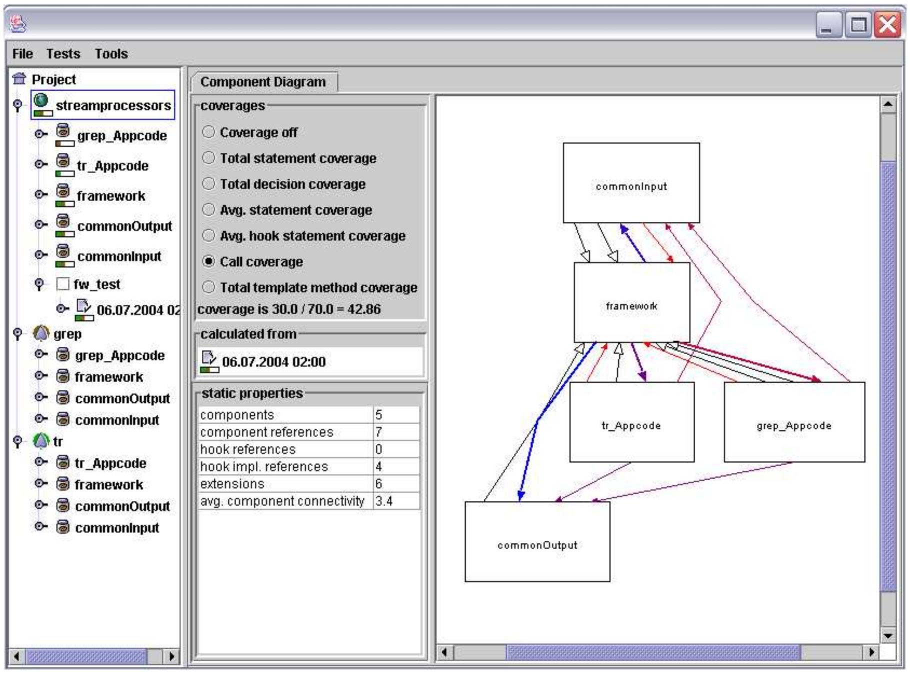

Figure 9: Product family views of RITA contain the common and the product-specific software in a diagram.

The application testers benefit from the family views by selecting to calculate the coverage in the framework from the coverages achieved in all testing performed in the family. This gives a more precise estimate of the test adequacy in the framework than merely calculating the coverage in the single application at hand. This is most useful when application testers try to minimize the testing of the framework done in application engineering. For example, the tester of the tr-application could analyze the statement coverage of Figure 9 to evaluate if testing of the framework is needed.

The domain testers can use RITA and the family views to achieve testing *by generalization*; that is, the correctness of the framework is tested and the testing is evaluated through the real products using it rather than isolating it with stubs. This is most beneficial during the evolution of the family stemming from a change in the framework. Also, the test coverage in the family scale can be improved most when the framework and its adaptation interface are better covered. These testing objectives are more suited to domain testing.

Looking at Figure 9, the domain tester can conclude that the testing of the framework is inadequate using the three lines of common test data. This follows as the test criterion of covering the virtually bound relations of the framework and its adaptations (the thick arrows starting from `framework`) is not met. This implies that the integration of framework and the adapting adapting software has not been fully exercised in all applications of the family during testing.

We have only described the most important features in RITA as when this paper is being written, RITA has well over hundred included services. RITA has 19 visual or textual presentations and 77 static measures. The basic coverages include statement, decision, method call, hook and template coverages. Due to changes in the calculation scope and variance between averages and totals in the grouping rules, RITA contains a total of 60 coverage measures. To complete the environment, we have a service providing a JUnit test execution environment [GB99], an RDBMS-based repository service, model management for the Java-language, and a group of various services that assist in coverage measurement. Together, the views and operations give seamless tool support for integration testing over the structure of the product family. The navigation, the views, and the operations in RITA provide an extensive exploration, measurement, test execution, and analysis of testing environment for framework-based product families.

RITA provides experimental support for domain and product testers in solving their problems. We have also given examples of ways to fulfill the requirements discussed in Section 3. RITA handles the scale, component-based testing, automated configuration management and test execution, management of commonality and variability, and testing of framework-based structures. The set of product family views and their operations are useful especially in test design and execution in domain engineering. The analysis and measurement of testing performed in the whole family guides the product tester in striving for adequacy with minimized effort.

6 Conclusions and future work

Product family and framework testing differ from testing of a single product mostly because of changes caused by moving towards the product family engineering approach. The division of domain and application engineering and the commonalities in the products requires dividing testing responsibilities between testing teams, applying incremental and regression testing, and synchronization and management of the asynchronous and distributed testing processes.

We have designed RITA to support selected tasks in integration testing of product families and frameworks. The implemented collection of services is useful in planning, execution, and analysis of integration testing. We have also implemented special testing services in the product family and framework contexts. Currently, RITA contains functionality enabling working with the commonality and variability of a product family and the polymorphism in framework-based software.

Another contribution of RITA is the incorporation of the model, the operations on it, and their integration in the user interface. We are not aware of any other testing tools that would integrate the model and the operations on so many levels of abstraction. The metrics and the visualizations on an architectural level are an especially interesting feature because they are informative in the design and analysis of component-based integration testing.

In the future, we would like to incorporate regression test selection methods, driver and stub generation services, automatic test case generation techniques, support in test asset evolution, and advanced automated configuration management into RITA. Based on the nature of the functionality in the missing services and our experience, the services can be defined and implemented using our model.

Another way to make RITA more general is to incorporate new implementations of the current service types. Our ideas include the introduction of new programming languages and multi-language support, table-driven testing, new views of the product family under test, and additional static and coverage measures.

A possible future direction of RITA is widening its scope toward general support for model-based software engineering by bringing functionality outside the testing domain into RITA. Maisa (Metrics for Analysis and Improvement of Software Architectures) is another software engineering tool that finds design patterns, calculates performance estimates, and other measures based on source code and various UML-diagrams [GPNV02]. By merging RITA and Maisa, we would create an open model-based software engineer's workbench.

RITA is currently a prototype. Many issues have to be overcome before it can be used in an industrial setting. Most problems are caused by the scalability of our current model and service implementations that keep the structure of the product family completely in memory. Other similar shortcuts have been taken in many places, and, before RITA can tackle testing of a large product family, robust implementations are needed.

In conclusion, we believe that the approach taken in RITA has enabled new and useful features especially in tool support for product family and framework testing but also in integration testing in general.

Acknowledgments

This work was funded by the ITEA project Families. The author thanks the other members of the project – Professors Jukka Paakki and Inkeri Verkamo, and co-researchers Raine Kauppinen and Juha Taina from the University of Helsinki – for the original research in the field and the valuable comments and ideas regarding this work.

References

[AO00] Roger T. Alexander and A. Jefferson Offutt. Criteria for testing polymorphic relationships. In *Proceedings of the 11th International Symposium on Software Reliability Engineering*, pages 15–23, 2000.

[Bin00] Robert V. Binder. *Testing Object-Oriented Systems*. Wiley, 2000.

[Bos00] Jan Bosch. *Design and Use of Software Architectures*. Addison-Wesley, 2000.

[CK99] Mei-Hwa Chen and Howard M. Kao. Testing object-oriented programs – an integrated approach. In *Proceedings of the 10th International Symposium on Software Reliability Engineering*, pages 73–82, 1999.

[Ecl04] Eclipse Foundation. *Eclipse home page*, 2004. URL: http://www.eclipse.org/ [July 2, 2004].

[ECM93] ECMA. Reference model for frameworks of software engineering environments. Technical Report TR–55, European Computer Manufacturers Association, June 1993.

[ER96] Nancy S. Eickelmann and Debra J. Richardson. An evaluation of software test environment architectures. In *Proceedings of the 18th International Conference on Software Engineering*, pages 353–364. IEEE Computer Society Press / ACM Press, 1996.

[FHB00] Mohamed E. Fayad, David S. Hamu, and Davide Brugali. Enterprise frameworks characteristics, criteria, and challanges. *Communications of the ACM*, 43(10):39–46, October 2000.

[GB99] Erich Gamma and Kent Beck. Junit: A cook's tour. *Java Report*, 4(5):27–38, May 1999.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of reusable object-oriented software*. Addison–Wesley, 1995.

[GPNV02] Juha Gustafsson, Jukka Paakki, Lilli Nenonen, and A. Inkeri Verkamo. Architecture-centric software evolution by software metrics and design patterns. In *Proceedings of the 6th European Conference on Software Maintenance and Reengineering (CSMR2002)*, pages 108–115, March 2002.

[Jaa02] Ari Jaaksi. Developing mobile browsers in a product line. *IEEE Software*, 19(4):73–80, July/August 2002.

[Kau03] Raine Kauppinen. Testing framework-based software product lines. Master's thesis, University of Helsinki, Department of Computer Science, 2003.

[KT03] Raine Kauppinen and Juha Taina. Rita environment for testing framework-based software product lines. In *Proceedings of the Eighth Symposium on Programming Languages and Software Tools*, pages 58–69, June 2003.

[McG01] John D. McGregor. Testing a software product line. Technical Report CMU/SEI-2001-TR-022, Software Engineering Institute, Carnegie Mellon University, December 2001.

[Mer03] Mercury Interactive. *Mercury Interactive TestDirector home page*, 2003. URL: http://www-svca.mercuryinteractive.com/products/testdirector/ [May 22, 2003].

[MvdH03] Henry Muccini and André van der Hoek. Towards testing product line architectures. In *Electronic Notes in Theoretical Computer Science*, volume 82. Elsevier, 2003.

[Nor02] Linda M. Northrop. SEI's software product line tenets. *IEEE Software*, 19(4):32–40, July/August 2002.

[Nor03] Linda M. Northrop. A framework for software product line practice – version 4.1. Technical report, Software Engineering Institute, Carnegie Mellon University, 2003. URL: http://www.sei.cmu.edu/plp/framework.html [November 11, 2003].

[Pre94] Wolfgang Pree. Meta patterns—a means for capturing the essentials of reusable object-oriented design. In *Object-Oriented Programming, Proceedings of the 8th European Conference ECOOP'94*, pages 150–162, Bologna, Italy, July 1994.

[Ran02] Charles Rankin. The software testing automation framework. *IBM Systems Journal*, 41(1):126–139, 2002.

[Szy98] Clemens A. Szyperski. *Component Software*. Addison-Wesley, 1998.

[Tev03] Antti Tevanlinna. Testauksen analysointityökalut ("Tools for analysis of testing", in Finnish). Master's thesis, University of Helsinki, Department of Computer Science, 2003.

[TTK04] Antti Tevanlinna, Juha Taina, and Raine Kauppinen. Product family testing – a survey. *ACM SIGSOFT Software Engineering Notes*, 29(2), March 2004.

[vdL02] Frank van der Linden. Software product families in Europe: The Esaps & Café Projects. *IEEE Software*, 19(4):41–49, July/August 2002.

[vGB01] Jilles van Gurp and Jan Bosch. Design, implementation and evolution of object oriented frameworks: concepts and guidelines. *Software – Practice and Experience*, 31(3):277–300, March 2001.

[Wey98] Elaine J. Weyuker. Testing component-based software: A cautionary tale. *IEEE Software*, 15(5):54–59, September 1998.