Electronic Notes in Theoretical Computer Science 82 No. 6 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume82.html 11 pages

Towards Testing Product Line Architectures

H. Muccini [1]

Dipartimento di Informatica
Universita' dell'Aquila
L'Aquila, Italy

A. van der Hoek [2]

Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, U.S.A.

Abstract

A product line architecture is a single specification capturing the overall architecture of a series of closely related products. Its structure consists of a set of mandatory elements and a set of variation points. Whereas mandatory elements are part of the architecture of every product in the product line architecture, variation points precisely define the dimensions along which the architectures of individual products differ from each other.

The increased use of product line architectures in today's software development projects poses several challenges for existing testing techniques. In this paper we discuss those challenges and discuss what we believe are opportunities for addressing them.

1 Introduction

Software testing consists of the "dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the specified behavior" [3]. Different overall strategies of software testing [26] exist, perhaps the most popular and widely-used being unit testing, integration testing, conformance testing, and regression testing [3]. All of these strategies share an important question upon which hinges their effectiveness in uncovering faults: which test cases to choose to perform the verification? In traditional approaches to software testing, this

[1] Email: muccini@di.univaq.it
[2] Email: andre@ics.uci.edu

question has typically been answered by using particular methodologies to select test cases based on the source code of the program to be tested [22].

With the advent and use of software architecture (SA) [17,25], source code no longer has to be the single source for selecting test cases. Instead, the abstraction provided by a SA allows testing to take place early and at a higher-level. This has two distinct advantages. First, it allows the detection of structural and behavioral problems before they are coded into the implementation simply by testing the architecture itself. For instance, it has been shown that the presence of performance problems or deadlocks can often already be detected at the architectural level [2,1]. Second, assuming a faithful preservation of the architecture in its implementation, additional information captured at the architectural level can guide the selection of test cases [5]. For example, precisely-defined component behaviors and constraints have been shown to be useful in regression testing to reduce the number of test cases that actually need to be considered.

While the use and adoption of SAs in industry has been moderately successful, the true benefit of using SAs (i.e., reuse) comes when they are applied in the form of product line architectures [6,19,20]. Rather than specifying a single architecture for a single software system, a product line architecture precisely captures, in a single specification, the overall architecture of a suite of closely-related products [6]. The techniques for doing so are rooted in the disciplines of SA and configuration management, and focus on a distinction between mandatory elements (which are present in the architecture of each and every product) and variation points (which define the dimensions along which the architectures of the individual products differ from each other) [7,14,11,12]. Variation points typically are specified either as optional elements, which may or may not be present in a product, and variant elements, which must be present in a product architecture but can be chosen to be one of a number of different alternatives (i.e., a component that represents a GUI variation point could be a Windows GUI component in one product architecture and a Unix GUI component in another product architecture). A single product line architecture may have many variation points that are often orthogonal to each other. As a result, hundreds and sometimes thousands of product architectures can be formed by a single product line architecture [15].

The use of a product line architecture brings both challenges and opportunities to the field of software testing. Challenges arise in the form of a new structure that must be tested: how to deal with optional elements or with the magnitude of products that may be present? Opportunities arise because a single product line architecture defines many similar products. As a result, for instance, the specification of a product line architecture can be leveraged for regression testing or a single test case may test multiple different variants.

The goal of this paper is to highlight the challenges and opportunities for software testing of product line architectures. Below, we first introduce the notion of a product line architecture. We then discuss how unit testing, integration testing, conformance testing, and regression testing apply in a setting of product line architecture-based software development. We conclude with some overall observations and a sketch of our future work.

2 Product Line Architecture

Software architectures provide high-level abstractions for representing the structure, behavior, and key properties of a software system [17]. These abstractions involve: *i*) descriptions of the elements from which systems are built, *ii*) interactions among those elements, *iii*) patterns that guide their composition, and *iv*) constraints on those patterns. In general, a system is defined as a set of *components*, their interconnections (*connectors*), and the overall organization of the components and connectors (*configuration*).

Whereas a "regular" architecture defines the structure of a single product, a product line architecture (PLA) defines the common architecture for a set of related products [6]. A PLA explicitly specifies: *i*) elements that are present in all products, *ii*) elements that are *optional*, and *iii*) elements that are always present but may be incorporated in one of many forms (*variants*). Specific *product architectures* are selected by choosing the desired optional components and selecting one component per variant. Perry [21] outlined the space of possibilities for modeling PLAs and observed that a PLA modeling technique must be both generic enough to encompass all members of a product line and specific enough to provide developers with adequate support for instantiating and implementing specific product architectures. While it is technically possible to reuse architectural styles for this purpose [25], experience with PLAs has shown a need for higher-level support in terms of explicit facilities for modeling optionality and variability [20].

To date, many *architecture description languages* (ADLs) have been developed to aid architecture-based development [17]. ADLs provide formal notations to describe software systems and are usually accompanied by various tools for parsing, analysis, simulation, and code generation of the modeled systems. Examples of ADLs include C2SADEL, Darwin, Rapide, UniCon, and Wright [16,17]. A number of these ADLs also provide extensive support for modeling behaviors and constraints on the properties of components and connectors [17], which can be leveraged to ensure the consistency of an architecture (e.g., by establishing conformance between the services of interacting components). These approaches have been extended to provide mechanisms to capture product line architectures. xADL 2.0 [8], Koala [20], and Mae [12] are examples of such product line architecture description languages.

Figure 1 shows a simple example product line architecture consisting of four components (connectors are omitted for simplicity). The component `foo` is a standard part of every architecture, as is the component `bar`. The component `bar`, however, is a variant component that can be instantiated in one of three different forms. The component `foobar` is an optional component, and

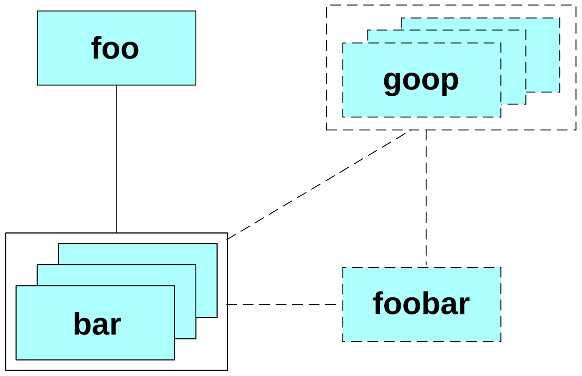

Fig. 1. Example Product Line Architecture

the component `goop` is an optional variant component. A surprising number of product architectures can be formed out of this product line architecture. In its simplest form, the product architecture consists of the component `foo` combined with one of the variants of component `bar`. A more complex product architecture can incorporate the component `foobar` and a variant of the component `goop`. In total, twenty-four different product architectures can be formed.

3 Testing Product Line Architectures

New testing techniques are needed to be able to test product line architectures. A first option could be to build new, ad hoc techniques from scratch. However, since PLAs and regular architectures conceptually share many commonalities, we believe that existing mechanisms with which SAs are tested can be adapted to PLAs.

In particular, we believe the following three architectural testing techniques [23,5,18]form the basis for our approach. In [23], architecture-based testing criteria are identified in order to cover certain elements of a software architecture. The approach, as applied to the Chemical Abstract Machine (CHAM) [13] formal ADL, initially defines the structures to be covered (i.e., *data*, *processing* and *connecting*), then specifies a set of paths covering those elements, and finally defines those inputs that cause those paths to be simulated. In [5], both behavioral and structural information is used to extract functional test cases. Component behaviors are modeled by the use of state-based models and a global model is obtained combining together those state machines. Relevant behavioral test cases are extracted and ran on the source code. Finally, in [18] a SA-based regression testing approach is proposed, based on an adaptation of traditional code-based selective regression testing techniques.

Below, we discuss how these techniques can be leveraged for the purpose of testing PLAs. We do so by examining how each of the existing SA-based techniques have covered the traditional testing activities of unit testing, integration testing, conformance testing, regression testing, and functional testing, as well as by discussing how these results may be extended to PLAs.

3.1 Unit testing

Unit testing checks each module for the presence of bugs. At the SA-level, unit testing checks whether each architectural component behaves according to its requirements. Since each component is used in building the overall SA, *each of them must be unit tested.* Existing SA-based testing approaches assume that components have been previously unit tested and do not provide SA-specific unit testing techniques.

At the PLA level, all components should be unit tested as well, including each optional component and each variant of a variant component. However, the order in which they have to be tested can be adjusted based on "priority". Standard components typically have highest priority, since they will be used in every PA. Similarly, optional and variant components that are used most often should have a higher priority than other optional or variant components that are rarely used. In this way, we can prioritize the initial effort involved in testing a PLA. Thus, we recommend a unit testing strategy in which standard components are unit tested first while variant and optional components are tested based on their level of usage in the overall PLA.

3.2 Integration testing

"Integration testing is the process of verifying the interaction between system components (possibly, and hopefully, already tested in isolation)" [3]. The basic strategy of integration testing is to bring together a set of components and test the behavior of the set as a whole. The integration of a system can be tested incrementally or using a big-bang approach.

For a normal SA, components and connectors are combined together according to the configuration of the architecture and integration testing is applied to the selected set of components and connectors. Examples of SA-based testing techniques are [5,24].

For a PLA, however, there is a difference since no single architectural configuration exists according to which the components and connectors should be incorporated. Instead, one has to follow an iterative path in which the addition of one component leads to the testing of multiple (partial) product architectures since the component has to be tested in each of the configurations. Clearly, this is an expensive and possibly even unachievable goal.

At first, it therefore seems like integration testing can only be performed using a big bang approach in which the whole PLA is assembled, followed by the selection and testing of each individual product architecture. This reduces the effort as compared to an iterative build-up approach, but it also limits the ability to pinpoint problems when they occur. A better solution would be to leverage the structure and nature of the elements in a PLA. First, one integrates the complete core of the PLA and uses a traditional approach to integration testing in doing so. Then, based on the observation that at least the core works properly, one can incorporate the other elements using the big-bang approach described for testing testing each of the product architectures.

A variant of this approach is presented in [15], which uses simple heuristics to test only particular combinations of optional and variant element. This helps in reducing the problem of combinatorial explosion. We believe other, similar heuristic approaches, must be developed and combined with the "core-first" approach to effectively perform integration testing of PLAs.

3.3 Conformance testing

Conformance testing is directed to demonstrate conformance to required capabilities. It is used to check the system correctness with respect to its requirements or the implementation conformance to a specification.

Given a SA, conformance testing has been used to detect conformance errors between the SA and its implementation [5]. The SA specification has been used as a reference model to which the source code should conform. The main problem with this kind of conformance testing is the necessity of a common model that makes it possible to compare the expected behavior of an SA with its real implementation.

Conformance testing can still be used with product lines. However, the picture is quite complicated since there is no one-to-one mapping between each PA and a separate implementation. For instance, a single implementation may realize one or more variants, but not all. Alternatively, there may be a different implementation for each variant. A tester, thus, must be aware of and exploit an explicit mapping between the PLA and its overall implementation. Based on this consideration, we can consider two options for defining conformance in PLAs: an implementation conforms to a PLA when it conforms to a single PA or when it conforms to all the possible PAs out of the PLA. Naturally, the second option is stronger than the first one but it is at the same time less realistic. A third option could consider the concept of "sufficient" conformance if an implementation conforms, at least, to all the constraints and functionalities associated to the mandatory, core elements of the PLA. In any of the three cases, conformance testing requires careful comparison of code execution with the architectural definition.

Conformance testing at the PLA level could also be used to test the conformance of each PA with respect to its PLA. This conformance is devoted to check that a PA conforms to all those structural and behavioral constraints imposed by the PLA. For example, reusing Figure 1, the PLA specification could impose that `foobar` can be used only in combination with one variant of the `goop` component. In this case, the PLA represents the specification while the PA represents one of its possible implementations.

3.4 Regression testing

Regression testing can be used during development or maintenance [9]. During development, regression testing is used to test families of similar products. During maintenance, it "attempts to validate modified software and ensure that no new errors are introduced into previously tested code" [10].

At the SA-level, regression testing can only be used during maintenance [18]. If a new version P2 of an implementation P1 is produced, regression testing techniques can be used to test the conformance of P2 to the initial architecture by reusing test cases generated to test P1's conformance to the same architecture. If a new version (SA2) of the architecture (SA1) is produced, SA2 test cases may be selected reusing SA1's test cases.

At the PLA-level, regression testing can be extensively applied during development and maintenance. During development, we can analyze two options: *i*) assuming that a PA in a given PLA has been tested, we can generally reuse a subset of those testing results in order to test another PA in the same PLA (option *v* in Figure 2). As in traditional regression testing, this happens if a test case covers only components common to both PAs. *ii*) If a program P1 has been conformance tested with respect to PA1 (a product architecture in a given PLA), then another program P2 can be conformance tested with respect to PA2 by selecting all those architectural test cases common to both architectures (option *vi* in Figure 2).

During maintenance, newer versions (P') of a program P can be produced (Figure 2, option iv). A code-based regression testing technique can be applied in order to test P' reusing results from P. An architecture-based regression conformance testing technique can be applied in order to test the conformance of P' with respect to its PA by reusing test cases previously selected to test P with respect to the same PA.

A final place in which regression testing plays an important role in PLAs is in their evolution. Continuously, new products are added, existing products are modified, and old products are retired. In managing such an evolving PLA structure, regression testing plays a crucial role in reducing the effort that is involved in testing the changing PLA. In particular, regression testing can be used to only test those parts of the PLA that are either changed itself or affected by the change. Clearly, suitable techniques are needed to determine the desired set of test cases.

3.5 The general picture

Summarizing what we said in this section, testing a PLA is more complex than testing a single SA. Figure 2 helps to summarize the testing activities related to SAs and PLAs. Dealing with a single SA, we can unit test the components (Figure 2.*i*), apply an integration testing strategy when components and connectors are integrated (Figure 2.*ii*), test the conformance of a possible implementation with respect to the SA specification (Figure 2.*iii*), and use

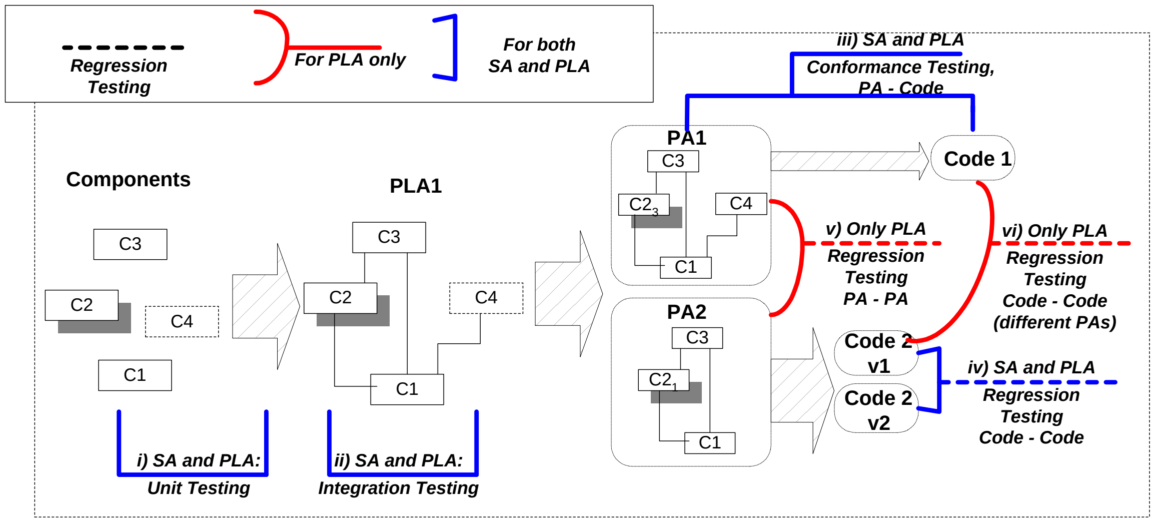
Regression Testing
For PLA only
For both SA and PLA
iii) SA and PLA
Conformance Testing, PA - Code
Components
PLA1
PA1
PA2
C3
C2
C4
C1
Code 1
Code 2 v1
Code 2 v2
v) Only PLA Regression Testing PA - PA
vi) Only PLA Regression Testing Code - Code (different PAs)
iv) SA and PLA Regression Testing Code - Code
i) SA and PLA: Unit Testing
ii) SA and PLA: Integration Testing

Fig. 2. Testing a PLA

regression testing for maintenance reasons (Figure 2.*iv*).

In testing a PLA, we need to keep in mind that (a) the topological description of a PLA is always incomplete (due to the presence of optional and variant elements), (b) PLA-level decisions are reflected into the PAs selected from the PLA, (c) many PAs can be extracted from the PLA, and (d) many implementations can be produced for each PA in the PLA. Unit, integration, conformance, and regression testing techniques can still be applied, but must be adapted and specialized for PLAs.

Analyzing Figure 2.*i* to Figure 2.*vi* we conclude that, at the PLA level:

i Unit testing needs to distinguish among standard, optional and variant components;

ii Integration testing needs to consider two different levels of integration: the overall PLA configuration and the individual PAs;

iii Conformance testing of a PA and its code can reuse information produced at the PLA level;

iv Regression testing two different implementations of the same product architecture can be realized by applying techniques already proposed for SA-based regression testing [18];

v The information produced by testing a PA in the PLA can be reused in order to test other PAs, using a development-level regression testing technique;

vi Information used to test the implementation of a certain PA in the PLA can be reused in order to test the conformance of another implementation with respect to its PA.

4 Conclusions and Future Work

The emergence of product line architectures provides some serious challenges for the field of software architecture. Perhaps the most daunting of those challenges is the need to test many closely related products that all are part of a single specification. That specification, at the same time, may also hold the key to successfully answering those challenges: by leveraging the commonality among the specified set of products and building upon the detailed information captured in a product line architecture, significant opportunities arise in adapting existing testing techniques to be able to address the testing of product line architectures.

At the forefront of our efforts is regression testing. In particular, we are exploring how existing regression test techniques can be applied to product line architectures specified in the xADL 2.0 [8] product line architecture description language. In addition, we are exploring how we can enhance that language with constructs that will make it easier to perform the testing strategies laid out in this paper.

References

[1] Allen, R., and D. Garlan. A Case Study in Architectural Modeling: The AEGIS System. In Proc. IWSSD eight, 1996.

[2] Aquilani, F., Balsamo, S., and P. Inverardi. *Performance analysis at the software architectural design level.* In Performance Evaluation 2001, N. 45, pp. 147-178.

[3] Bertolino, A. "Knowledge Area Description of Software Testing". In SWEBOK: The Guide to the Software Engineering Body of Knowledge, Joint IEEE-ACM Software Engineering.

[4] Bertolino, A., and P. Inverardi. Architecture-based software testing. In Proc. ISAW96, October 1996.

[5] Bertolino, A., Inverardi, P., and H. Muccini. An Explorative Journey from Architectural Tests Definition downto Code Tests Execution. In IEEE Proc. Int. Conf. on Software Engineering (ICSE2001), pp. 211-220, May 2001.

[6] Bosch, J. "Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach". Addison Wesley, 2000.

[7] Bosch, J., Florijn, G., Greefhorst, D., Kuusela, J., Obbink, J.H., and K. Pohl. Variability Issues in Software Product Lines. In Proc. of the fourth Software Product-Family Engineering Workshop (PFE), October 2001, pp. 13-21. LNCS 2290.

[8] Dashofy, E.M., van der Hoek, A., and R.N. Taylor. An Infrastructure for the Rapid Development of XML-Based Architecture Description Languages. In Proc. of the 24th Int. Conf. on Software Engineering, 2002.

[9] Harrold, M.J. Architecture-Based Regression Testing of Evolving Systems. In Proc. Int. Workshop on the ROle of Software Architecture in TEsting and Analysis (ROSATEA), CNR-NSF, pp. 73-77, July 1998.

[10] Harrold, M.J. Testing: A Roadmap. In A. Finkelstein (Ed.), ACM ICSE 2000, The Future of Software Engineering, pp. 61-72, 2000.

[11] van der Hoek, A. Capturing Product Line Architectures. In Proc. of the 4th Int. Software Architecture Workshop, 2000.

[12] van der Hoek, A., Mikic-Rakic, M., Roshandel, R., and N. Medvidovic. Taming Architectural Evolution. In Proc. of the Sixth European Software Engineering Conference and the Ninth ACM SIGSOFT Symposium on the Foundations of Software Engineering, 2001, pp. 1-10.

[13] Inverardi, P., and A.L. Wolf. *Formal Specifications and Analysis of Software Architectures Using the Chemical Abstract Machine Model.* IEEE Trans. on Software Engineering, **21, 4** (April 1995), pp. 100-114.

[14] Kang, K., Cohen, S., Hess, J., Nowak, W., and S. Perterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Software Engineering Institute, 1990.

[15] McGregor, J.D. Testing a Software Product Line. Technical Report, CMU/SEI-2001-TR-022, ESC-TR-2001-022.

[16] Medvidovic, N., and R.N. Taylor. A Framework for Classifying and Comparing Architecture Description Languages. In Proc. ESEC/FSE'97, LNCS vol. 1301, Sept. 1997.

[17] Medvidovic, N., and R.N. Taylor. *A Classification and Comparison Framework for Software Architecture Description Languages.* IEEE Transactions on Software Engineering, **26(1)** (2000), pp. 70-93.

[18] Muccini, H., and D.J. Richardson. Software Architecture-based Regression Testing. Internal Report, University of California, Irvine.

[19] Northrop, L.M. Reuse That Pays: ICSE Keynote Presentation. In Proc. of the 23rd Int. Conf. on Software Engineering, 2001.

[20] van Ommering, R., van der Linden, F., Kramer, J., and J. Magee. *The Koala Component Model for Consumer Electronics Software.* Computer, **33(3)** (2000), pp. 78-85.

[21] Perry, D.E. Generic Descriptions for Product Line Architectures. In Proc. of the Second Int. Workshop on Development and Evolution of Software Architectures for Product Families (ARES II), 1998.

[22] Rapps, S., and E.J. Weyuker. *Selecting Software Test Data Using Data Flow Information.* IEEE Trans. on Software Engineering, **SE-11** (1985), pp. 367-375.

[23] Richardson, D.J., and A.L. Wolf. Software Testing at the Architectural Level. In Proc. Second Int. Software Architecture Workshop (ISAW-2), pp. 68-71, October 1996.

[24] Richardson, D.J., Stafford, J., and A.L. Wolf. A Formal Approach to Architecture-based Software Testing. Technical Report, University of California, Irvine, 1998.

[25] Shaw, M., and D. Garlan. "Software Architecture: Perspectives on an Emerging Discipline". Prentice-Hall, 1996.

[26] Zhu, H., Hall, P.A.V., and J.H.R. May. *Software Unit Test Coverage and Adequacy.* ACM Computing Surveys, **29, 4** (Dec. 1997), pp. 366-427.