Assume-Guarantee Testing of Evolving Software Product Line Architectures

Maurice H. ter Beek[1], Henry Muccini[2], and Patrizio Pelliccione[2]

[1] ISTI–CNR, Pisa, Italy
maurice.terbeek@isti.cnr.it
[2] Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica
Università dell'Aquila, Italy
{henry.muccini,patrizio.pelliccione}@univaq.it

Abstract. Despite some work on testing software product lines, maintaining the quality of products when a software product line evolves is still an open problem. In this paper, we propose a novel assume-guarantee testing approach as a solution to the following research question: *how can we verify the correct functioning of products of an software product line when core components evolve?* The underlying idea is to retest only some of the products that conform to the software product line architecture and to infer, using assume-guarantee reasoning, the correctness of the other products. Assume-guarantee reasoning moreover permits the retesting of only those components that are affected by the changes.

Keywords: Assume-guarantee testing, Evolving software product lines, Software testing, Compositional verification.

1 Introduction

Software product line engineering makes use of different components to describe and realize families of systems, such as requirements, architectural and design models, and implementation components [1, 2]. The architecture of a software product line is typically referred to as a software product line architecture and it is meant to define the common reference architecture for the products that are related to a specific family. Variability is achieved by identifying variation points as places in the product line architecture where specific decisions are reduced to a choice among several *features*, but the feature to be chosen for a particular product variant is left open (due to optional, mandatory or alternative features).

For enterprises from safety-critical domains, such as avionics, the application of software product line engineering technology is often problematic because of the high costs of certification efforts. In such domains, every product has to pass a costly certification stage, even if it belongs to an established family of products for which certification efforts have already been performed. It is therefore important to reduce the effort needed to retest modified products. By predicting the impact of evolution of an software product line, we might be able to avoid re-certification of certified products that have evolved. This cannot

P. Avgeriou (Ed.): SERENE 2012, LNCS 7527, pp. 91–105, 2012.

solve the issue of repeated certification in a product line altogether, since testing one product of a family in general does not provide any guarantee for another product of the family. A possible solution, however, is to investigate how testing and variability can be combined to selectively test only components actually affected by the evolution.

In [3], we proposed a first step towards a solution to this problem by reusing, adapting and combining state-of-the-art techniques. We found assume-guarantee reasoning well suited for evolving systems. The environment of a component is seen as a set of properties, called *assumptions*, that should be satisfied for it to function. If these assumptions are satisfied by the environment, then components in this environment will typically satisfy other properties, called *guarantees*. By appropriately combining the assume and guarantee properties, it is possible to prove the correctness of an entire system before actually constructing it.

The idea we presented was thus to annotate components of a product line architecture with pairs of assume-guarantee properties, considering a component's environment as the composition of the remaining components. In this way, we enabled compositional verification based on assume-guarantee reasoning to deal with evolution in product line architectures. The underlying idea is to decompose a system specification into (asssume-guarantee) properties that describe the behavior of a system's subset, to model check these properties locally, and to deduce from the local checks that the complete system satisfies the overall specification. The main advantage is that one never has to compose all subsystems, thus avoiding the state explosion problem. On the downside, assume-guarantee verification by means of model checking cannot scale to the size of industrial systems (as remarked in [4]).

For this reason, we envision the use of software testing for verifying evolving product line architectures. While many approaches to validate software product lines and their products in a cost effective way by exploiting similarities among products and using proper variability management have been proposed [5–15], we know of only a few approaches that investigate how to retest the resilience of products when a software product line evolves [16, 17].

After analyzing state-of-the-art techniques for verifying and validating software product lines, in this paper we present a preliminary approach to retest the resilience of products of a software product line by properly extending and adapting assume-guarantee reasoning to evolving software product lines. The main goal is to permit selective (re)testing of assume-guarantee properties on only those components and products of the software product line that are actually affected by the evolution.

Section 2 recalls the problem that drives our research effort. Section 3 presents background information on assume-guarantee reasoning and assume-guarantee testing. Section 4 presents our proposed solution for assume-guarantee testing of evolving software product lines. Section 5 briefly reports on related work on regression testing of software product lines, while Section 6 concludes the paper outlining some future work on our wish list.

2 Problem Setup

We recall the research problem that we set up in [3]: how to guarantee the correct evolution of a software product line upon changes in components of its underlying product line architecture. We illustrate the problem in Fig. 1, based on the following example that is originally due to Paul Clements et al. at the Software Engineering Institute:

> *"I run the same software in different kinds of helicopters. When the software in a helicopter powers up, it checks a hardware register to see what kind of helicopter it is and starts behaving appropriately for that kind of helicopter. When I make a change to the software, I would like to flight test it only on one helicopter, and prove or (more likely, assert with high confidence) that it will run correctly on the other helicopters. I know I can't achieve this for all changes, but I would like to do it where possible."*

In a software product line context, this example can be re-phrased as: assuming various products (helicopters) have been derived from a software product line, which have moreover been formally certified, what can be concluded for new products obtained from the software product line by modifying one or more core components?

We assume that all products of the product line must be guaranteed to conform to a specific standard. Furthermore, we assume that there is a policy according to which any change to a core component requires all products containing that core component to be rebuilt. The question is whether it is necessary to re-validate all the products of the software product line or whether a subset can suffice. For instance, in the aforementioned example, when we change the software of the helicopter line, such as installing a new kind of radio across the fleet, we would like to flight test it only on one helicopter and assert with high confidence that it will run correctly on the other helicopters. In this paper we concentrate on modifications that are the result of changing, adding or removing components, but not their connections.

3 Assume-Guarantee Reasoning and Testing

Compositional verification is thus based on decomposing the system specification into a set of properties each of which describing the behavior of a system's subset. In general, checking local properties over subsystems does not imply the correctness of the entire system. This is due to the existence of mutual dependencies among components. More precisely, each single component cannot be considered in isolation but must be considered as behaving and interacting with its environment (i.e., the rest of the system).

Assume-guarantee reasoning is one of the most promising approaches proposed for compositional reasoning. It was originally introduced in the Ph.D. thesis of Cliff Jones [18] and in the context of temporal logic by Amir Pnueli [19]. Assume-guarantee reasoning considers both components and the environment

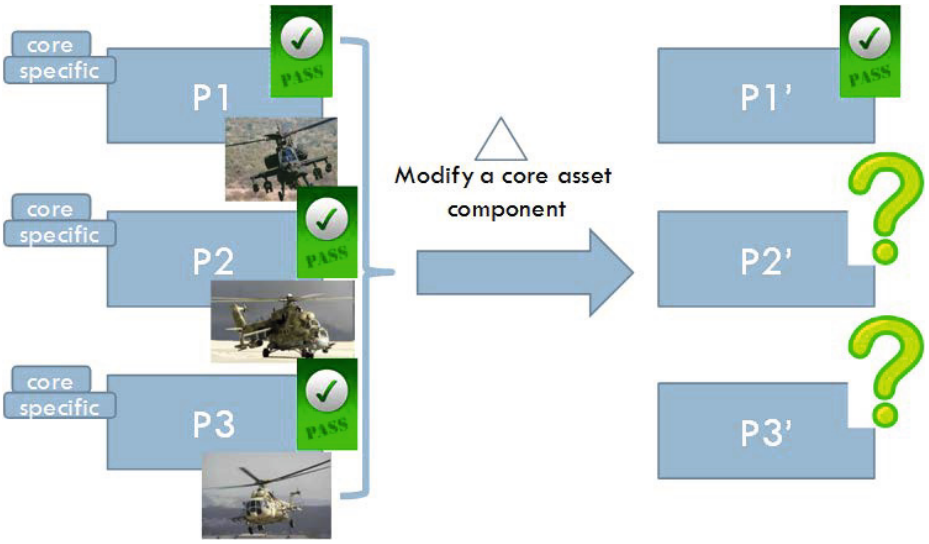

Fig. 1. The helicopter certification problem

they interact with. The environment is described a set of properties that should be satisfied for it to correctly interact with the components. These properties are the *assumptions* that the component makes on the environment. If they are satisfied by the environment, then the component behaving in this environment usually satisfies other properties, the *guarantees*. As said before, a system can sometimes be proved correct without actually constructing it through the appropriate combination of assume and guarantee properties.

We use Pnueli's notation to state that if the environment of component M satisfies logic formula φ (i.e., φ is the *assumption* that M makes about the components it interacts with), then in that environment M satisfies ψ (i.e., M *guarantees* ψ):

$$\langle \varphi \rangle \; M \; \langle \psi \rangle$$

Pnueli's classical reasoning chain then becomes:

$$\frac{\langle\,\rangle \; M \; \langle \varphi \rangle \qquad \langle \varphi \rangle \; M' \; \langle \psi \rangle}{\langle\,\rangle \; M \cdot M' \; \langle \psi \rangle}$$

in which '$\cdot$' is a suitable composition operator. This reasoning chain should be interpreted as follows: if M, with no assumption on its environment, satisfies φ, and M', over an environment that satisfies φ, satisfies ψ, then without any assumption on its environment $M \cdot M'$ satisfies ψ. In this paper, we consider M and M' to model component behaviors and φ and ψ to be formulae expressed in Linear Temporal Logic.

As witnessed by [20], the main difficulties of applying assume-guarantee reasoning are (i) *generating* the assumptions and (ii) *decomposing* a system into subsystems with the purpose of efficiently verifying general system properties. Therefore, in [21] three main dimensions to be considered when dealing with assume-guarantee reasoning were identified:

1. The *composition operator* should be carefully selected and has to be associative. It defines a *system and properties decomposition* and this is fundamental in order to correctly work with the reasoning chain;
2. The *assumptions generation* technique is fundamental for assuring effective assume-guarantee reasoning;
3. The language used to specify the system and that used to specify the assume-guarantee properties are crucial. Semantic relationships between these two languages (if they differ) are fundamental to make the assumption generation process fully automatic.

The authors of [22] introduce a framework for performing assume-guarantee reasoning in an incremental and fully automatic fashion. More specifically, the approach automatically generates via a learning algorithm assumptions that the environment needs to satisfy for the property to hold. These assumptions are initially approximate, but become gradually more precise by means of counterexamples obtained by model checking the component and its environment. In [23], the authors observe that in reality a component is only required to satisfy properties in specific environments. Inspired by these motivations, they generate assumptions that characterize exactly those environments in which the component satisfies its required property.

Assume-guarantee testing has been proposed as a technique to complement assume-guarantee model checking, to be used for the verification of components' implementations [4]. While formal verification with assume-guarantee testing increases scalability when compared to traditional formal verification techniques such as model checking, it may still be unfeasible when applied to the implementation of large-scale industrial systems. For this reason, Giannakopoulou et al. proposed in [4] an approach that, after applying assume-guarantee formal verification at the design level, uses assume-guarantee information to test whether the components' implementations continue to satisfy the assumptions. In affirmative cases, this will show that there is no incompatibility between the design models and their implementations. The main advantages of assume-guarantee testing rely on the possibility of detecting violations with a higher probability than in the case of traditional testing, avoiding state explosion problems that typically arise when combining components, and with the same coverage as that of traditional systems — but with fewer tests.

More formally, consider a system S in a possibly empty context E. Let S be composed of n components $C_1, C_2, \ldots, C_n$[1]. The idea is to produce for each C_i, with $1 < i < n$, both G_i and A_i, representing respectively the local requirements that C_i has to guarantee and the assumptions that characterize the context

[1] Here the term component is used as a synonym of part.

in that point. In other words, each component C_i will guarantee property G_i whenever its context satisfies assumption A_i.

Next consider two components C_1 and C_2 (but the reasoning can straightforwardly be extended to a multitude of components) and two implementations I_1 and I_2 of these components, respectively. After having validated with assume-guarantee model checking that $C_1 \cdot C_2$ satisfies a property ψ, we subsequently want to test whether $I_1 \cdot I_2$ still satisfies ψ. The idea for doing so is to check this separately, via unit testing, for the two component implementations $\langle\varphi\rangle\ I_1\ \langle\psi\rangle$ and $\langle\ \rangle\ I_2\ \langle\varphi\rangle$. Assume-guarantee testing is then run according to the following three steps:

1. The assumption φ is used to restrict the execution of the component I_1 when producing test traces. This means that sets of test traces T_1 and T_2 are produced from I_1 composed with an implementation of φ, and from I_2 composed with an implementation of the universal environment, respectively;
2. The resulting traces are individually checked against the appropriate assume-guarantee premise. Thus, each trace in T_1 is checked against ψ, and each trace in T_2 is checked against φ, individually, without requiring the construction of all the interleavings of the two components' implementations;
3. If either of these above checks fails, then there is an incompatibility between the components' models and their implementations that needs to be fixed. Otherwise, $I_1 \cdot I_2 \models \psi$.

Assume-guarantee testing is still testing, i.e. it lacks exhaustive coverage. However, as said before, assume-guarantee testing has the potential of checking more system behaviors with the same amount of coverage as traditional testing.

4 Assume-Guarantee Testing of Evolving Software Product Line Architectures

While we have seen that a lot of research efforts have been devoted to software product line testing [17, 24–26, 13], limited research has been conducted on how to test evolving software product lines (some research that has been carried out is discussed in Section 5). The solution we propose below combines the principles of traditional regression testing, in the context of evolving software product line architectures, with the use of assume-guarantee reasoning. As the architecture of the software product line plays a central role in our approach, the latter can thus be considered architecture-centric.

Figure 2 contextualizes our work: all components used in the original product line architecture of the software product line of interest (i.e., A, B, C, D and E) are enriched with assume and guarantee properties. These assumptions can automatically be calculated by extending the approach presented in [23] to software product line architectures.

The configuration of the product line architecture (i.e., the way components are instantiated, selected, and connected) provides context to the assumptions

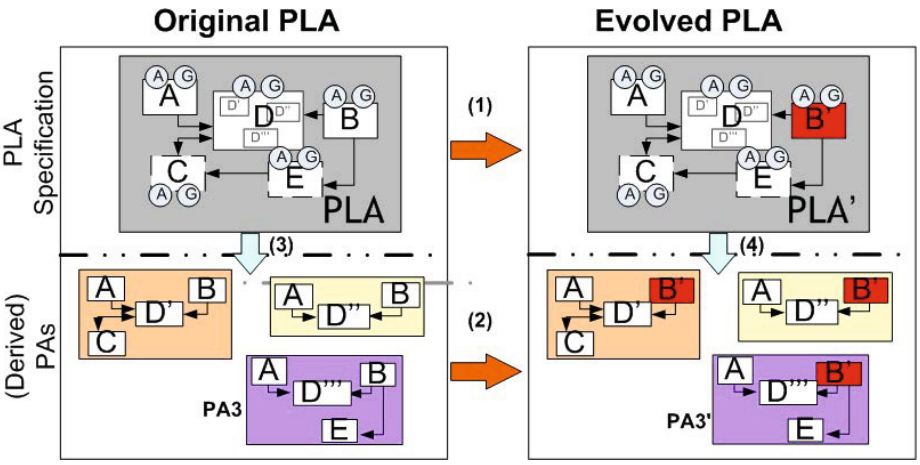

Fig. 2. The original software product line architecture (PLA) and the evolved one, including the product architectures (PAs) of their derived products

and guarantees: given a component C with its assume-guarantee pair, its environment becomes the subarchitecture connected with C. What is challenging in assume-guarantee reasoning in the context of product line architectures is that the reasoning is performed on the product line architecture, including all variation points, rather than on each single concrete product — or better, on the architecture of each single product. This means that the assumptions will have to be calculated in a smart way, taking into account the commonalities and variability among the components.

Once a component evolves, this modification is expected to have an impact on a number of product architectures, namely each one that contains the modified component. For instance, consider that component B evolves into B' (cf. Fig. 2). Then the assume-guarantee pairs of both B and B' will have to be checked.

In such a context, our solution envisions a combination of regression testing and assume-guarantee testing in the context of evolving product line architectures. Conceptually, this requires us to better understand two issues.

First, we need to understand how to extend the assume-guarantee testing approach proposed in [4] to evolving (product line) architectures. In [4], in fact, the assume-guarantee pair associated to each component in the architecture is used to generate component-specific testing traces. These traces, when adequately evaluated against the appropriate assume-guarantee premise, can demonstrate whether the composition of components can produce failures. Assuming that a product (line) architecture has been tested, the challenge becomes how to apply assume-guarantee reasoning for regression testing of the modified product (line) architecture (cf. (1) and (2) in Fig. 2).

Second, we need to understand how to use the relationships between a product line architecture and its derived product architectures (cf. (3) in Fig. 2) in order

to be able to apply regression testing to the evolved product architectures (cf. (4) in Fig. 2).

Summarizing, we envision what we call a *double regression testing* approach, in which an evolved product (e.g., PA3′ in Fig. 2) can be tested not only based on how it regressed from an original product (i.e., PA3 in Fig. 2) but also based on its relationship with the architecture of its product family (i.e., PLA′ in Fig. 2).

The remainder of this section is organized as follows. In Section 4.1, we present the assume-guarantee testing approach from [4] adapted and extended to work with product line architectures, while Section 4.2 describes the strategy followed to select the product architectures that — once retested — can minimize the retesting of other product architectures.

4.1 Applying Assume-Guarantee Testing to Evolving Product Line Architectures

In this section, we describe how to adapt the assume-guarantee testing approach presented in [4] (and recalled in Section 3) to evolving product line architectures. We initially explain how to adapt the assume-guarantee testing approach to evolving architectures, and subsequently how this approach can be extended to evolving product line architectures.

Assume-Guarantee Testing Applied to Evolving Systems

Consider a system S that is decomposed into n components $C_1, C_2, \ldots, C_n$. Assume that for each C_i, with $1 < i < n$, a pair of assumptions A_i and guarantees G_i is available. As said before, the assumptions can automatically be calculated by a suitable assumption generation function defined over the considered language (e.g., the $\mathcal{L}^*$ learning algorithm defined in [23]).

Now set C_i as the component that is affected by a change and that is substituted by C_x. According to the approach presented in [21], the changes' correctness can be checked locally over the changed component(s). In case component C_x can substitute C_i without consequences (i.e., the assume-guarantee pairs A_x, G_x and A_i, G_i match), then the properties guaranteed by C_i are also guaranteed by C_x and no retesting is needed. This case provides an extraordinary advantage with respect to traditional (non compositional) verification approaches, where local changes have to be checked against the entire system. On the other hand, when component C_x cannot substitute C_i without consequences (i.e., the assume-guarantee pairs A_x, G_x and A_i, G_i do not match), then the assume-guarantee reasoning chain my help understand the effect of a change.

In fact, since the components composing the system S are organized as a chain of assumptions and guarantees (through which the composition of the system is realized), assume-guarantee reasoning enables an immediate understanding of how local changes affect the entire system, without applying traditional impact analysis approaches. Again, this facilitates (and potentially automates) the regression testing analysis of evolving systems.

Assume-Guarantee Testing Applied to Evolving Product Line Architectures

In order to extend the assume-guarantee regression testing approach outlined in Section 4.1 to product line architectures, it becomes particularly important to consider the commonalities and variability in product line architectures and, more specifically, normal, optional and variant components. Below we present in detail the way in which the approach of Section 4.1 has to be reconsidered in the case of such components.

Normal or Optional Component. When a component C has to be substituted with a component C', there are two possible cases:

1. The pair of assumptions and guarantees of C' matches the assumptions and guarantees of C and thus no re-verification is needed.
2. The pair of assumptions and guarantees of C' does not match that of C. This may impact only a part or, in the worst case, the entire chain. To measure the impact, we check whether an environment exists for C' such that the guarantee of C can be satisfied. If it does, then the chain allows to check if the component that should match this assumption with its guarantee is analyzed. This reasoning is iterated until each incongruence in the assume-guarantee chain has been resolved. If, on the other hand, no such environment exists for C', then we have to also analyze the right side of the chain.

Variant Component. Let D be an abstract variant component with n variants $D_1, D_2, \ldots, D_n$. When a variant D_i evolves in D_i', a reasoning similar to the one made for normal and optional components must be performed. In addition, however, we have to consider that D_i was a variant of D and therefore also D_i' should be a variant of D. This means that the pair of assumptions and guarantees of D should hold also for D_i'. If this is indeed the case, then the chain in which D is involved does not require changes. Otherwise, even the assumptions and guarantees of D have to be updated and the changes must be propagated by suitably recalculating all involved assume and guarantee properties in the chain.

Adding or removing a component in a product line architecture requires a reasoning similar to the one we just outlined for substitution. We can have an optimal integration or removal, meaning that no assumptions and guarantees must be recalculated (such as the removal of an optional component). In general, however, assumptions and guarantees must be recalculated (and in the worst case, the entire chain must be recalculated).

The next step is to apply the assume-guarantee testing approach. When applying assume-guarantee testing in the context of evolving product line architectures, the assume-guarantee testing process presented in [4] has to be applied as follows: we initially apply assume-guarantee reasoning at the level of the product line architecture in order to analyze the conformance of product architectures to assume-guarantee properties. Successively, we apply assume-guarantee testing to check the conformance of the implementation of the product architecture with respect to the its specification (exactly as proposed in [4]).

When the product line architecture evolves, we first need to identify the product architectures that should be retested. As we will explain in Section 4.2, it is important to appropriately select the product architectures to be retested since the selection strategy can impact the number of tests to be performed on other product architectures. Once the product architectures to be retested have been selected, we have to apply the assume-guarantee strategy described above to reassure the conformance of each such product architecture to the modified product line architecture (or to evolve the product architecture in accordance to the changes). Subsequently, we again apply assume-guarantee testing to check the conformance of the implementation of the product architecture with respect to its specification (exactly as proposed in [4]). Finally, we need to reselect a subset of test traces for those components that need to be retested. Existing test traces for the component implementation have to be modified only when the assume-guarantee properties of its component specification C have changed. New test traces have to be added for all the new components added to the product architecture.

As far as automation is concerned, in this paper we consider both the architectural models and the implementations to be represented (at different levels of abstraction) by labeled transition systems. As a result, the Labelled Transition System Analyser[2] as described in [23] becomes an ideal candidate tool to be extended in order to cope with test traces generation. Such a setting is inherited from [4] and can be applied when detailed models are constructed through the refinement of more abstract models.

4.2 Testing Strategy: Selection of Products to Retest

The main strategic decision we need to make in order to apply the proposed double regression testing approach is the choice of the products that need to be retested and the test sets. The apparently most useful information we have at hand is the so-called feature model of the software product line, the original product (line) architecture and the previous tests. By making intelligent use of the commonalities and variability inherent to the software product line, we can select both the products that need retesting and a set of test cases from the existing ones. Ideally, this allows us to avoid having to retest all products and to rerun all test cases.

A *feature model* has become the de facto standard variability model in software product line engineering. It provides a compact representation of all products of a product family in terms of their features, and additional constraints among them. Graphically, features are represented as the nodes of a tree, with the family as its root and relations between these features representing constraints. The first three relations below form the tree, while the latter two model additional constraints:

[2] http://www.doc.ic.ac.uk/ltsa/

Optional features may (but need not) be present only if their parent is present;
Mandatory features are (have to be) present if and only if their parent is present;
Alternative features are such that only one is present if their parent is present;
Requires is a unidirectional relation indicating that the presence of one feature requires that of the other;
Excludes is a bidirectional relation indicating the presence of two features to be mutually exclusive.

From a feature model we can thus extract all necessary information on the relationships between features, such as which features exclude each other and which are optional. By analyzing the features that are involved in the component that has evolved, we can obtain useful information on the impact on other features. From the product line architecture we know which product architecture contains the component that has evolved. Combining this information, we can thus select the product architecture that needs to be retested and appropriately adjust the test sets. In a similar way, in [27] it is shown that the addition/modification of so-called *conservative* and *regulative* features in a software product line requires only a subset of the new products to be model checked.

To make this more concrete, consider once more Fig. 2. Component B has evolved in component B' and this component is part of three product architectures, among which PA3. Now we inspect the feature model regarding the features that are present in B and B', and in particular regarding their difference. If, for instance, B' contains a feature f that B did not, then we need to inspect the feature model for the features that are related with f. If the feature model states that the inclusion of f in a product *excludes* the presence of a feature g, for instance, then we need to inspect the product architectures of which B' is part to see whether or not the other components contain g. Moreover, we need to add the exclusion of g to the assumptions that B' makes on the environment (in addition to the assumptions inherited from B). Similar reasonings can easily be imagined in case B' contains less features than B or exactly the same features as B. Likewise, a related reasoning applies to the cases in which the feature model states that f *requires* a feature g and/or other relationships.

5 Related Work

Software product line testing consists of using product line artifacts (e.g., requirements, architectures, code with variability) or artifacts of products derived from a software product line, to select test suites enabling the validation of a software product line and of its derivable products. During software product line testing at least two main (and opposite) dimensions need to be considered: testing the core components present at the domain engineering level (i.e., testing the software product line artifacts) and testing the products derivable from the product line during application engineering (i.e., testing the product that is part of a software product line).

Testing product lines during domain engineering means testing the software product line based on all the artifacts common to the product line (e.g., software product line testing based on domain-level requirements, design, and realized components). The opposite dimension means testing the single products that can be obtained during application engineering. At this stage, the main goal is to test the variety of products obtained by assembling core and domain-specific components, possibly written in different programming languages, distributed across the network and executed in various platforms. Other techniques use a mix of product and product line information in order to derive a testing campaign (as discussed in the systematic study in [17]).

Several software product line testing approaches have been proposed so far in the literature, many of which use software product line requirements (with explicit variability modeling) for selecting requirements-based test suites, while most of the approaches use models (of the requirements or of the architecture) defined formally or semi-formally for the derivation of test cases. A problem common to all approaches is the number of test cases to consider, which obviously increases exponentially with the number of features of the software product line (cf. [13] for an overview and comparison of several scalable testing techniques that aim at reducing the number of products to be tested, among which so-called combinatorial interaction testing).

Since the focus of this paper is on evolving software product lines, the work most closely related to our research is that on software product line regression testing, which aims to minimize the effort to retest a software product line when components change. The most advanced regression testing approach for software product line architectures can be found in [16], in which three different software product line evolution scenarios and a regression testing approach for software product lines are described. By considering a regression testing strategy RT that takes as input two versions of the same component, three scenarios can be defined, as follows:

1. Given a reference architecture RA, once a core component in RA is changed, a strategy RT(RA,RA′) is needed for retesting the reference architecture;
2. Given a reference architecture RA and a product P derived from it, a strategy RT(RA,P) can be used to test P under the assumption that RA has already been tested, taking advantage of the similarities between the RA and P;
3. Given two products P1 and P2, both derived from the same reference architecture RA, a strategy RT(P1,P2) can be applied to test P2 under the assumption that P1 has already been tested, taking advantage of the similarities among products derived from the same RA.

By taking the two versions (the original and its modified one) as the main input of the RT regression testing approach, a 4-step approach (from regression testing planning to test reporting) is proposed. Architectural specifications, feature models, the product map, and the feature dependency diagram can be used to identify portions that need retesting.

The preliminary work of Engström in [28] presents observations coming from two systematic reviews of the relevant literature; one deals with regression test

selection and the other one with software product line testing. As future work, Engström plans to investigate (among others) how to perform regression testing in software product lines.

6 Conclusions and Future Work

The most important concepts characterizing software product line engineering as a discipline for the development of a diversity of software products or systems based on the underlying architecture of the product platform, are a product line's commonalities and variability (often defined in terms of features whose relations are expressed in a feature model). Commonalities define what different software products or systems have in common and guides the production of domain-specific core components. Variability defines the ability to change or customize a software product or system (i.e., to distinguish one product from another in the software product line).

In this paper, we define a testing approach to exploit the commonalities and variability of the products of a software product line when the software product line (architecture) is subject to evolution. The idea is to retest only selected products while inferring the "correct" functioning of other products conforming to the software product line. Our approach, heavily based on that of [4], makes use of assume-guarantee reasoning, which is used both to verify the (underlying architectural) design of the product or system and to drive the testing phase.

While we are obviously aware that this work is quite preliminary, we believe it can trigger interesting discussions, and for this purpose we are submitting it to this workshop. As a consequence, there is a long list of future work we would like to accomplish in the near future.

First, we plan to apply the presented approach to the Arcade Game Maker Pedagogical Product Line whose specification and code are available online[3].

Second, we want to experiment with the presented approach also on other industrial case studies, like the one presented in Section 2.

Some further real-world industrial case studies can be found through the Architecture Support for Testing initiative[4] which this research is part of.

Finally, we ideally would like to automate our approach. As said before, labeled transition systems are currently used for describing the components' behavior as well as the assume and guarantee properties of a given software product or system, while in the approach of [4] the aforementioned Labelled Transition System Analyser is used for automatically generating assumptions. Our approach, however, would require an extension in order to be able to deal with product line architectures, for which we intend to move from labeled transition systems to so-called modal transition systems, which were recognized in [11] as a useful formal model for describing in a compact way the possible operational behavior of all products of a product line. As a result, we foresee the need for an appropriate extension of the Modal Transition System Analyser [29] (a tool

[3] http://www.sei.cmu.edu/productlines/ppl/
[4] http://www.henrymuccini.com/index.php?pageId=AST

built on top of the Labelled Transition System Analyser in order to deal with limited variability) for automatically generating assumptions for product line architectures, and for coping with test traces generation.

References

1. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley (2002)
2. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer (2005)
3. ter Beek, M.H., Muccini, H., Pelliccione, P.: Guaranteeing Correct Evolution of Software Product Lines: Setting up the Problem. In: Troubitsyna, E.A. (ed.) SERENE 2011. LNCS, vol. 6968, pp. 100–105. Springer, Heidelberg (2011)
4. Giannakopoulou, D., Pasareanu, C., Blundell, C.: Assume-guarantee testing for software components. IET Softw. 2(6), 547–562 (2008)
5. Asirelli, P., ter Beek, M.H., Fantechi, A., Gnesi, S.: Formal Description of Variability in Product Families. In: SPLC 2011 Conference Proceedings, pp. 130–139. IEEE Press (2011)
6. Asirelli, P., ter Beek, M.H., Fantechi, A., Gnesi, S., Mazzanti, F.: Design and Validation of Variability in Product Lines. In: PLEASE 2011 Workshop Proceedings, pp. 25–30. ACM Press (2011)
7. ter Beek, M.H., Mazzanti, F., Sulova, A.: VMC: A Tool for Product Variability Analysis. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 450–454. Springer, Heidelberg (2012)
8. Classen, A., Cordy, M., Heymans, P., Legay, A., Schobbens, P.Y.: Model Checking Software Product Lines with SNIP. To appear in Int. J. Softw. Tools Technol. Transfer (2012)
9. Classen, A., Heymans, P., Schobbens, P.Y., Legay, A.: Symbolic model checking of software product lines. In: ICSE 2011 Conference Proceedings, pp. 321–330. ACM Press (2011)
10. Classen, A., Heymans, P., Schobbens, P.Y., Legay, A., Raskin, J.F.: Model checking lots of systems: efficient verification of temporal properties in software product lines. In: ICSE 2010 Conference Proceedings, pp. 335–344. ACM Press (2010)
11. Fischbein, D., Uchitel, S., Braberman, V.A.: A foundation for behavioural conformance in software product line architectures. In: ROSATEA 2006 Workshop Proceedings, pp. 39–48. ACM Press (2006)
12. Lauenroth, K., Pohl, K., Toehning, S.: Model Checking of Domain Artifacts in Product Line Engineering. In: ASE 2009 Conference Proceedings, pp. 269–280. IEEE Press (2009)
13. Perrouin, G., Oster, S., Sen, S., Klein, J., Baudry, B., le Traon, Y.: Pairwise Testing for Software Product Lines: Comparison of Two Approaches. To appear in Software Qual. J. (2012)
14. Schaefer, I., Gurov, D., Soleimanifard, S.: Compositional Algorithmic Verification of Software Product Lines. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 184–203. Springer, Heidelberg (2011)
15. Thüm, T., Schaefer, I., Kuhlemann, M., Apel, S.: Proof Composition for Deductive Verification of Software Product Lines. In: ICSTW 2011 Conference Proceedings, pp. 270–277. IEEE Press (2011)

16. da Mota Silveira Neto, P.A.: A Regression Testing Approach for Software Product Lines Architectures: Selecting an efficient and effective set of test cases. LAP (2010)
17. da Mota Silveira Neto, P.A., do Carmo Machado, I., McGregor, J.D., de Almeida, E.S., de Lemos Meira, S.R.: A systematic mapping study of software product lines testing. Inf. Softw. Technol. 53(5), 407–423 (2011)
18. Jones, C.B.: Development Methods for Computer Programs including a Notion of Interference. PhD thesis, Oxford University (1981)
19. Pnueli, A.: In Transition from Global to Modular Temporal Reasoning about Programs. In: Logics and Models of Concurrent Systems, pp. 123–144. Springer (1985)
20. Cobleigh, J.M., Avrunin, G.S., Clarke, L.A.: Breaking up is hard to do: An evaluation of automated assume-guarantee reasoning. ACM Trans. Softw. Eng. Methodol. 17(2), 1–52 (2008)
21. Inverardi, P., Pelliccione, P., Tivoli, M.: Towards an assume-guarantee theory for adaptable systems. In: SEAMS 2009 Workshop Proceedings, pp. 106–115 (2009)
22. Cobleigh, J.M., Giannakopoulou, D., Păsăreanu, C.S.: Learning Assumptions for Compositional Verification. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 331–346. Springer, Heidelberg (2003)
23. Giannakopoulou, D., Pasareanu, C.S., Barringer, H.: Component Verification with Automatically Generated Assumptions. Autom. Softw. Eng. 12(3), 297–320 (2005)
24. Engström, E., Runeson, P.: Software product line testing: A systematic mapping study. Inf. Softw. Technol. 53(1), 2–13 (2011)
25. Kim, C.H.P., Batory, D.S., Khurshid, S.: Reducing combinatorics in testing product lines. In: AOSD 2011 Conference Proceedings, pp. 57–68. ACM Press (2011)
26. Kim, C.H.P., Bodden, E., Batory, D., Khurshid, S.: Reducing Configurations to Monitor in a Software Product Line. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G., Roşu, G., Sokolsky, O., Tillmann, N. (eds.) RV 2010. LNCS, vol. 6418, pp. 285–299. Springer, Heidelberg (2010)
27. Cordy, M., Classen, A., Schobbens, P.Y., Heymans, P., Legay, A.: Managing evolution in software product lines: a model-checking perspective. In: VaMoS 2012 Workshop Proceedings, pp. 183–191. ACM Press (2012)
28. Engström, E.: Regression Test Selection and Product Line System Testing. In: ICST 2010 Conference Proceedings, pp. 512–515. IEEE Press (2010)
29. D'Ippolito, N., Fischbein, D., Chechik, M., Uchitel, S.: MTSA: The modal transition system analyser. In: ASE 2008 Conference Proceedings, pp. 475–476. IEEE Press (2008)