Software ENgineering

Composing configurable Java components

T. van der Storm

REPORT SEN-R0503 JANUARY 2005

Copyright © 2005, Stichting Centrum voor Wiskunde en Informatica
P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-369X

Composing configurable Java components

ABSTRACT
This paper presents techniques to reason about the composition of configurable components and to automatically derive consistent compositions. The reasoning is achieved by describing components in a formal component description language, that allows the description of component variability, dependencies and configuration actions. It also enables the automatic, configuration-driven, derivation of product instances. To illustrate the approach we instantiate the abstract component model for Java components (packages).

2000 Mathematics Subject Classification: 68N30, 68N01
1998 ACM Computing Classification System: D.2.13, D.2.9, D.2.7, D.2.12
Keywords and Phrases: variability; component-based software engineering; variation management; configuration; feature diagrams; product population; composition; aspect-oriented programming
Note: This work was sponsored in part by the Netherlands Organisation for Scientific Research NWO, Jacquard project Deliver.

Composing Configurable Java Components

Tijs van der Storm
Centrum voor Wiskunde en Informatica (CWI),
Kruislaan 413, NL-1098 SJ Amsterdam, The Netherlands,
storm@cwi.nl

CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

25th January 2005

ABSTRACT
This paper presents techniques to reason about the composition of configurable components and to automatically derive consistent compositions. The reasoning is achieved by describing components in a formal component description language, that allows the description of component variability, dependencies and configuration actions. It also enables the automatic, configuration-driven, derivation of product instances. To illustrate the approach we instantiate the abstract component model for Java components (packages).

1998 ACM Computing Classification System: D.2.13, D.2.9, D.2.7, D.2.12

Keywords and Phrases: Java, variability, component based software engineering, software variation management, configuration, feature diagrams, product population, composition, aspect-oriented programming

Note: This work was sponsored in part by the Netherlands Organisation for Scientific Research NWO, Jacquard project *Deliver*.

1. Introduction

The dream of composing software systems out of reusable components is as old as the field of software engineering itself. The notion of *variability* is closely related to the concept of reuse. If the variable parts of a software system can be identified, the common parts can be reused, thus shortening time-to-market and increasing customer satisfaction.

The concept of variability is a central topic in research on *product lines*. A product line contains different systems, all implemented using the same set of reusable software assets. Component-based software engineering is a way of implementing product lines. By letting multiple components implement the same interface, different implementations can be chosen for the same feature. Different component compositions result in different software products. Components are then the units of configuration.

However, there is a saying “maximizing reuse, minimizes use” [SGM02]. This dictum is based on the assumption that maximally reusable components are by implication maximally small. But if we can *configure* components for reuse in different contexts, this assumption does not necessarily hold. Such *configurable components* need not be tiny at all. They are, in fact, a different approach to reuse: instead of focussing on factorization (decomposing a system in smaller and smaller pieces), the focus is on malleability (allowing lightweight adaptation of components for use in different contexts).

Composing systems out of configurable components raises questions concerning the relation between variability and composition. A choice has to be made about when components are configured and when they are composed, and in what order. Another question is how to bind configuration parameters to implementations. While there is much work on implementing variability (see, e.g., [AG01]), the verified instantiation of products using configurable components has received little attention.

In this paper we describe how configurable Java components can be composed to automatically derive valid product instances. Components are described in a component description language formal enough to be able to verify the consistency of component variability and configuration. It is the starting point for instantiating products by composing components and binding of features. To implement the variability described in configuration interfaces, we use a combination of Aspect Oriented Programming (AOP) [KLM$^+$97], and Java property files.

This paper is organized as follows. In Section 2, we provide some background to configurable components. We survey the concepts of product families, product lines, and product populations, and discuss how features are bound to implementations in the case of Java components. Then, in Section 2, we present a language for component descriptions. We describe an example product population that is subsequently used to illustrate the notation. In Section 3, we discuss the consistency requirements for configuration and composition how these requirements enable automated product instantiation. We show how component descriptions are mapped to configuration and composition actions. Finally we present related work and future work.

2. Configurable Components

2.1 Background

The notion of variability is important as it is the dual of software reuse. The idea is that, if we can identify what is common to different product variants, and what is variable, the common parts can be reused in all product variants. A commonality and variability analysis can have a great impact on the architecture of a system, since a good modularization allows the variation of components, modules or subsystems through information hiding. The units of decomposition (components, modules, subsystems etc.) thus become the units of variation.

However, there are cases, when components as the only units of variation cannot be used for implementing variability. There are many examples of features that cannot adequately be decomposed in any unit of decomposition. For some of these feature a simple compile-time conditional (e.g. using the C preprocessor) is enough. In other cases more advanced techniques, such as Aspect Oriented Programming (AOP) [KLM$^+$97], must be applied to obtain variation at the level of components (e.g. a component with logging support, or without). Common to all these cases is that variability occurs at the level of components themselves, instead of at the level of the composition.

To further motivate the idea of configurable components, we put it in the context of three common approaches to variability. The following hierarchy of approaches to variability is by no means meant to be definitive. Its purpose is primarily to tentatively give clear definitions to concepts concerning variability. Of course, the borders between the approaches are fuzzy and not absolute.

The oldest approach to variability is the *product family* approach. In a product family, many *variations* of the *same* system are derived from a set of software components. Configuration tends to be fine-grained. Keywords such as tailoring, customization, platform/protocol/format variation come to mind. The primary driver is to increase customer satisfaction. Examples of product families are large ERP software suites [JBB04] and the Linux kernel. Both examples exist in many variations, yet each variation fulfills more or less the same requirements.

The *product line* approach goes one step further than the product family approach. In this case, many *different systems* are derived from the same set of software components. For product lines, the concept of *architecture* is crucial since the units of variation are usually more coarse-grained. An example of a product line is the package base developed at CWI. From this set of packages different toolsets for language analysis are derived: ASF+SDF Meta-Environment, ApiGen, JTom, XT. Each toolset fulfills different requirements, yet (re)uses the same components.

The most elaborate form of reuse is obtained in the *product population* approach. Product populations contain many *different products* in many *different variations*. In fact, the approach can be seen as the union of product family and product line approach: each product in the product line is a product family on its own. Product populations are more rare than product families and product lines. The canonical example is the product population developed by Van Ommering in the field of consumer electronics [vO00]. A single set of components supports different products (software for DVD players, TVs, Videorecorders etc.) *and* different variants of these products (for different types of DVD players, TVs, Videorecorders etc.).

The product population approach is the natural environment for configurable components. Using configurable components, different products can be derived by composing components differently. Because each component exposes variability, such compositions are by implication configurable as well. Thus, each composition represents a product family. In other words: each product family is a composition of *component families*.

2.2 Properties, Aspects, and Dependencies

In the previous subsection we presented a bird's eye view on configurable components. In this subsection, we zoom in and discuss the features of configurable components in the context of this paper.

A basic java component is represented by a java *package*. Packages contain classes and interfaces that form the exported interface of the component. In Java, a package always corresponds to a directory. The composition of packages is performed by feeding the appropriate directories including the compiled sources into the standard `jar` tool.

To enable reasoning about the configuration and composition of java components, these components have to be described in a formal component description language. Descriptions in this language, consist of a *configuration interface* and *binding interface*. The configuration interface models the configuration space of a component. The process of selecting features is called the *configuration task*. When a consistent set of features is selected, the composition algorithm executes the appropriate configuration actions, defined in the binding interface. The result is a `jar`-file ready for deployment.

The configuration interface of a component communicates the component's *variability* to the outside world (clients of the component). The binding interface, on the other hand, describes the *realization of variability* local to the component. Realizing variability amounts to binding *variation points* (parameters) to certain *implementation variants*. Our model supports two realization techniques:

- *Properties* Some variation points need to be bound at runtime. One way of implementing this is by checking for a certain property in the component's property-file. A selected feature could thus result in the setting of some property. Combining this technique with the Abstract Factory **??** design pattern enables choosing between different implementations of the same interface.
- *Aspects* Aspect Oriented Programming (AOP) [KLM$^+$97] is targeted at cross cutting features (features that resist conventional modularization). *Aspects* encapsulate this kind of features. Integrating an aspect with the source code of the component (weaving) can be seen as a technique for realizing variability, since there is always the choice *not* to weave.

Usually components cannot be deployed in isolation, that is, a component usually has *dependencies*. Component descriptions can contain references to other components as part of their binding interface. The binding interface thus also functions as a *requires* interface. If a component (the *client* component) depends on another component (the dependency component), it may pass configuration parameters to it. This way, a component can (partially) configure the component it depends on.

3. Component Description

3.1 A Small Tree Transformation Product Population

In order to evaluate our component description language, we first present tiny product population to illustrate the concept. We have actually implemented this product population, and used it to instantiate multiple products.

Consider developing a small application for transforming trees. This application consists of a number of different components. First there is a `transformer` component, providing a number of primitives to transform and/or analyze tree structured data. An optional feature of this transformation component is the tracing of the transformation process.

Another component provides the classes for the required tree data structure. This `tree` component contains two different implementations for the same `Tree` interface. In addition to the choice between different implementations, trees can optionally be enhanced with a `Visitable` interface. This allows the trees to be used together with yet another component, a generic visitor/traversal framework (implementing the Visitor design pattern [GHJV86]).

The transformation component obviously has a dependency on the tree component, but it also requires the traversal functionality. If the tree component is required to provide the traversal feature, it should in turn depend on the traversal component. The choice among tree implementations, however is left to the user of the transformation component. In other words: the some of the variability of the tree component propagates to the transformation component.

Why do these three components represent a product population? There are two reasons for this. Firstly, both the `tree` component and the `traversal` component can be (re)used independently of the `transform` component. That is, they can be reused in products other than the transformation application. Secondly, the all components described here, except the traversal component, are configurable. That is, both `tree` and `transformer` represent a component family. If different systems are composed using these configurable components, these different systems become, by implication, configurable themselves. Therefore: each type of system is a product family as well.

3.2 Component Description Language

Our component description language is has a syntax similar to Java. A component description consists of two parts. The first part is the configuration interface. This interface declares how a component should be configured. The second part consists of configuration and composition *actions*, which specify how to bind variation points to implementation variants. An example of the notation is displayed in Figure 1. The example represents the term rewriting product population described in the introduction.

```
package transformer {
  Transformer: tracing?
  if (tracing) { weave(Tracing); }
  require(tree, traversal);
}

package tree {
  Tree: all(one-of(list,array), traversal?)
  if (array)  { treefactory = array.ArrayTreeFactory; }
  if (list) { treefactory = list.ListTreeFactory; }
  if (traversal) {
    require(traversals);
    weave(TreeTraversal);
    if (list) { weave(list.ListTreeTraversal); }
    if (array) { weave(array.ArrayTreeTraversal); }
  }
}

package traversals { }
```

Figure 1: Description of the tree transformation product population

The configuration interface of component consists of a *feature description* [vDK02]. Feature descriptions are textual analogs of feature diagrams. Features can be defined to mandatory (all), alternative (one-of), inclusive (more-of), or optional (?). Features that cannot be decomposed any further are atomic features (lowercase featurenames). Finally, a feature description may declare additional constraints on atomic features (not displayed in the example). An example of such a constraint would be that some atomic feature *a* requires or excludes the presence of aother atomic feature.

The action part consists of java-like conditionals and statements. Conditionals check whether a particular atomic feature is enabled or not. The statements bind variants to variation points by performing one or more of three actions:

- setting a property ($p = v$)

- weaving an aspect (weave(a))
- requiring another component (require(n, f^*))

When a property is set, the local property file of the current component is modified. The weaving of an aspect is enabled by adding the aspect to the argumentfile passed to Aspect/J. Finally, requiring another component induces a build-time dependency. This means that the classpath should be adjusted in order to find the required component. Required features can be passed to the dependency component.

As an example to the notation, Fig. 1 displays the description of the tree transformation population. It consists of three components: `transformer`, `tree` and `traversals`. The `transformer` component has one dependency on `tree` and it requires the traversal functionality to be present. The if-statement ensures that if the tracing feature is enabled, the `Tracing` aspect is weaved into the sources of `transformer`.

The `tree` component has a more elaborate configuration interface. It can be instantiated using different tree implementations: array- or list-based. These implementations are hidden with the Abstract Factory design pattern [GHJV86]. Depending on the chosen implementation variant, a property is set to indicate which concrete tree factory should be used.

Enabling the traversal feature, is implemented using a combination of aspects. This feature also induces a dependency on the `traversals` component, which cannot be configured any further.

In the next section we will describe how the consistency of these component description can be automatically checked, and how they are used for instantiating products.

4. Automating Product Instantiation

4.1 Consistency Requirements

Component descriptions should be consistent to prevent the derivation of invalid product instances. The component description language discussed in the previous section allows a number of checks to be automatically performed: well-formedness checks and consistency checks. We are currently working on a Feature Analysis and Manipulation Environment (FAME) which is a suitable platform for the implementation of these checks. FAME supports the controlled evolution, analysis, and querying of textual feature descriptions. We plan to extend this environment with support for the component description language described here.

Well-formedness The first consistency requirements is well-formedness. Non-well-formed component descriptions are not logically flawed, but can nevertheless lead to spurious product instantiations. Well-formedness applies to both configuration interfaces and binding interfaces.

Well-formedness of the configuration interface equals well-formedness of feature definitions. The feature description should not, for instance, include a feature expression like the following: `more-of(foo,foo)`. Another well-formedness violation is duplicate definitions for the same composite features. For the well-formedness of a configuration interface, we have implemented a typechecker which detects many kinds of violations.

The binding interface of component description should also be well-formed. This means that binding actions are not ambiguous. As an example, consider two alternative features `a` and `b`. Then, the following statement is not well-formed:

```
if (a) {
  if (b) {
    ...
  }
  ...
}
```

Although this kind of configuration conditionals is not logically inconsistent, they are meaningless, since `a` and `b` never occur together.

Well-formedness of binding actions can be checked as follows. Nested conditionals induce (sub)sets of feature selections. For the example above (`if (a) { if (b) {...}}`) the induced feature set is $\{a, b\}$.

We can use this feature set to check the Well-formedness of configuration and composition actions. For an action to be meaningful, such a feature set should be a subset of a consistent feature selection. Clearly, if feature `a` and `b` are alternative, they will not both be part of a consistent feature selection. In that case the action is not well-formed. For the `tree` component, for instance, the action `if (array) { if (list) {...}}` is not well-formed.

Configuration consistency The first consistency requirement is internal consistency of the configuration interface. An inconsistent configuration interface cannot be configured, and thus cannot be instantiated.. An example of an inconsistent feature description would be the following:

```
A: one-of(b, c)
b requires c
```

Feature `b` requires feature `c`, but they are defined to be alternative. This is a contradiction.

The second requirement is that any feature selection *used* to configure a component is valid with respect to the configuration interface. For all components depending (conditionally) on a component c, the union of features passed to c should not invalidate the configuration interface of c. Assume, for instance, that component c declares two atomic features to be alternative, say `a` and `b`. Then, if a certain component enables `a` and another component that is part of the same composition, enables `b` then this is a consistency violation.

We have described a method to check these logical consistency requirements elsewhere [vdS04]. That technique is based on translating component descriptions to Binary Decision Diagrams (BDDs) [Bry92]. A slightly different mapping of the component descriptions of this paper can be used to obtain the same results.

4.2 Composition and Binding

If component descriptions are consistent, they can be used to generate product instances by composing all components required for a certain (consistent) selection of features. To compose Java components means compiling all required source files and putting the result in a JAR-file. Assume we have a set of required components C, and a set of required features F that pass the consistency tests defined in the previous subsection. Then, a valid product instance is created by the following process:

- For each component $c \in C$
  - Add the sources of c to the set of all sources.
  - Obtain the actions of c that are by enabled F.
  - For each action:
    - weave(a): add aspect a to the set of all sources
    - $p = v$: set property p to v
- Compile the set of all sources using AspectJ
- Create a `jar`-file, including property files.

We plan to describe the composition and binding process using the example in an extended version of this paper.

5. CONCLUDING REMARKS

5.1 Related Work

Our work is greatly influenced by the KOALA component model, developed by Rob van Ommering [vOB02, vO01, vO00]. KOALA allows the description of configurable components. Configuration occurs by binding values or implementations to interfaces which may declare configuration parameters. A KOALA compiler then creates a configured C implementation. A drawback of the approach is the tight coupling between the mechanisms of variation and the C programming language. We have therefore separated the realization of the variability from the configuration interface description. This way, implementation details are completely hidden behind abstract feature descriptions, yet it is still possible to link configuration interface and binding interface in a meaningful way.

Another source of influence is package-based software development [dJ03]. In package-based software development a software system is decomposed in source packages. Package descriptions declare a list of build-time configuration flags and dependencies. Source-tree composition [dJ02] is then used to obtain a bundle that has a top-level build process. The binding of compile-time flags to implementations is completely hidden using the `autoconf` tool set. Thus it is not possible to check any consistency constraints automatically. Moreover, using a feature description as configuration interface is more expressive than just a list of options.

In [vDdJK02] the authors investigate how products can be instantiated using both source-tree composition and feature descriptions. However, these feature descriptions have a meaning at the level of the composition only. In our approach feature descriptions function both at the component-level and at the composition-level. Binding of variation points is implemented using reflection and customer factories. There is no direct mapping of feature sets to configuration actions.

Checking feature diagrams for consistency is an active area of research [vDK02, CBB+03, Man02] but the level of formality varies. The problem is that checking the consistency is equivalent to satisfiability, so most approaches do not scale. Our approach is based on BDDs [vdS04], which make the exponential feature space manageable.

5.2 Contribution

Configurable components increase the opportunities for reuse and product variation. However, they also present both conceptual and technical challenges. In this paper we have dealed with these challenges by describing components in a formal component description language.

The conceptual complexity of composing configurable components is ameliorated by automating formal reasoning about variability, configuration and composition. Component descriptions and compositions can be checked for consistency so that valid product instantiation is guaranteed.

Technical challenges include the question on how to realize variability and how to maintain a meaningful link between abstract configuration interface and concrete binding interface. We have shown how this can be accomplished for Java components, when feature binding is realized using property files and aspect oriented programming.

5.3 Future Work

We plan further research in four directions: tool support, further formalization, generalization and validation. Tool-support is needed for configuration and composition. Further formalization will help to understand and manage the complexity that is introduced by variability inheritance. It is worth investigating whether our component model could be made oblivious to implementation language. That is, the set of possible configuration action becomes a parameter of the component description language. Heterogeneous product populations could then be uniformly described using our component description language. Finally, to really assess the flexibility of the approach we will have to validate it in practice.

Acknowledgements Gratitude goes to Gerco Ballintijn and Slinger Jansen for valuable suggestions to improve drafts of this paper.

References

[AG01] Michalis Anastasopoulos and Cristina Gacek. Implementing product line variabilities. In *Proceedings of the 2001 Symposium on Software Reusability (SSR-01)*, volume 26, 3 of *SSR Record*, pages 109–117, New York, May 18–20 2001. ACMPress.

[Bry92] Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[CBB+03] Fei Cao, Barrett R. Bryant, Carol C. Burt, Zhisheng Huang, Rajeev R. Raje, Andrew M. Olson, and Mikhail Auguston. Automating feature-oriented domain analysis. In *Proc. of the International Conf. on Software Engineering Research and Practice (SERP'03)*, 2003.

[dJ02] M. de Jonge. Source tree composition. In Cristina Gacek, editor, *Proceedings: Seventh International Conf. on Software Reuse*, volume 2319 of *LNCS*, pages 17–32. Springer-Verlag, April 2002.

[dJ03] M. de Jonge. Package-based software development. In *Proc.: 29th Euromicro Conf.*, pages 76–85. IEEE Computer Society Press, 2003.

[GHJV86] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of reusable object-oriented software*. Addison–Wesley, 1986. ISBN: 0–201–63361–2.

[JBB04] Slinger Jansen, Gerco Ballintijn, and Sjaak Brinkkemper. Integrated SCM/PDM/CRM and Delivery of Software Products to 160.000 Customers. Technical report, CWI, 2004. Submitted for publication.

[KLM+97] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242, Berlin, Heidelberg, and New York, 1997. Springer-Verlag.

[Man02] Mike Mannion. Using first-order logic for product line model validation. In G. Chastek, editor, *Proc. of The 2nd Software Product Line Conf. (SPLC2)*, number 2379 in LNCS, pages 176–187, 2002.

[SGM02] Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 2nd edition, 2002.

[vDdJK02] A. van Deursen, M. de Jonge, and T. Kuipers. Feature-based product line instantiation using source-level packages. In G. J. Chastek, editor, *Proceedings: Second Software Product Line Conf. (SPLC2)*, number 2379 in LNCS, pages 217–234. Springer-Verlag, August 2002.

[vDK02] A. van Deursen and P. Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10(1):1–18, March 2002.

[vdS04] Tijs van der Storm. Variability and component composition. In Jan Bosch and Charles Krueger, editors, *Software Reuse: Methods, Techniques and Tools: 8th International Conference (ICSR-8)*, volume 3107 of *Lecture Notes in Computer Science*, pages 86–100. Springer, June 2004.

[vO00] Rob van Ommering. Mechanisms for handling diversity in a product population. In *Proc. of the 4th International Software Architecture Workshop (ISAW4)*, 2000.

[vO01] Rob van Ommering. Configuration management in component based product populations. In *Proc. of the 10th Intl. Workshop on Software Configuration Management*, May 2001.

[vOB02] Rob van Ommering and Jan Bosch. Widening the scope of software product lines: from variation to composition. In G. Chastek, editor, *Proc. of The 2nd Software Product Line Conf. (SPLC2)*, number 2379 in LNCS, 2002.