Variability and Component Composition*

Tijs van der Storm

CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
storm@cwi.nl

Abstract. In component-based product populations, variability has to be described at the component level to be able to benefit from a product family approach. As a consequence, composition of components becomes very complex. We describe how this complexity can be managed automatically. The concepts and techniques presented are the first step toward automated management of variability for web-based software delivery.

1 Introduction

Variability [13] is often considered at the level of one software product. In a product family approach different variants of one product are derived from a set of core assets. However, in component-based product *populations* [14] there is no single product: each component may be the entry-point for a certain software product (obtained through component composition).

To let this kind of software products benefit from the product family approach, we present formal component descriptions to express component variability. To manage the ensuing complexity of configuration and component composition, we present techniques to verify the consistency of these descriptions, so that the conditions for correct component composition are guaranteed.

This paper is structured as follows. In Sect. 2 we first discuss component-based product populations and why variability at the component-level is needed. Secondly, we propose a Software Knowledge Base (SKB) concept to provide some context to our work. We describe the requirements for a SKB and which kind of facts it is supposed to store. Section 3 is devoted to exploring the interaction of component-level variability with context dependencies. Section 4 presents the domain specific language CDL for the description of components with support for component-level variability. CDL will serve as a vehicle for the technical exposition of Sect. 5. The techniques in that section implement the consistency requirements that were identified in Sect. 2. Finally, we provide some concluding remarks and our future work.

* This work was sponsored in part by the dutch national research organization, NWO, Jacquard project DELIVER.

2 Towards Automated Management of Variability

Why Component Variability? Software components are units of independent production, acquisition, and deployment [9]. In a product family approach, different variants of one system are derived by combining components in different ways. In a component-based product population the notion of *one* system is absent. Many, if not all, components are released as individual products. To be able to gain from the product family approach in terms of reuse, variability must be interpreted as a component-level concept. This is motivated by two reasons:

- In component-based product populations no distinction is made between component and product.
- Components as unit of variation are not enough to realize all kinds of conceivable variability.

An example may further clarify why component variability is useful in product populations. Consider a component for representing syntax trees, called `Tree`. `Tree` has a number of features that can optionally be enabled. For instance, the component can be optimized according to specific requirements. If small memory footprint is a requirement, `Tree` can be configured to employ hash-consing to share equal subtrees. Following good design practices, this feature is factored out in a separate component, `Sharing`, which can be reused for objects other than syntax trees. Similarly, there is a component `Traversal` which implements generic algorithms for traversing tree-like data structures. Another feature might be the logging of debug information.

The first point to note, is that the components `Traversal` and `Sharing` are products in their own right since they can be used outside the scope of `Tree`. Nevertheless they are required for the operation of `Tree` depending on which variant of `Tree` is selected. Also, both `Traversal` and `Sharing` may have variable features in the very same way.

The second reason for component variability is that not all features of `Tree` can be factored out in component units. For example, the optional logging feature is strictly local to `Tree` and cannot be bound by composition.

The example shows that the variability of a component may have a close relation to component dependencies, and that each component may represent a whole family of (sub)systems.

The Software Knowledge Base The techniques presented in this paper are embedded in the context of an effort to automate component-based software delivery for product families, using a Software Knowledge Base (SKB). This SKB should enable the web-based configuration, delivery and upgrading of software. Since each customer may have her own specific set of requirements, the notion of variability plays a crucial role here.

The SKB is supposed to contain all relevant facts about all software components available in the population and the dependencies between them. Since we want to keep the possibility that components be configured before delivery, the

SKB is required to represent their variability. To raise the level of automation we want to explore the possibility of generating configuration related artifacts from the SKB:

Configurators Since customers have to configure the product they acquire, some kind of user interface is needed as a means of communication between customer and SKB. The output of a configurator is a selection of features.

Suites To effectively deliver product instantiations to customers, the SKB is used to bundle a configured component together with all its dependencies in a configuration suite that is suitable for deployment. The configuration suite represents an abstraction of component composition.

Crucial to the realization of these goals is the consistency of the delivered configurations. Since components are composed into configuration suites before delivery, it is necessary to characterize the relation between component variability and dependencies.

3 Degrees of Component Variability

A component may depend on other components. Such a client component requires the presence of another component or some variant thereof. A precondition for correct composition of components is that a dependent component supports the features that are required by the client component. Figure 1 depicts three possibilities of relating component variability and composition.

The first case is when there is no variability at all. A component C_a requires components $C_1, ..., C_n$. The component dependencies $C_1, ..., C_n$ should just be present somehow for the correct operation of C_a. The resulting system is the composition of C_a and $C_1, ..., C_n$, and all component dependencies that are transitively reachable from $C_1, ..., C_n$.

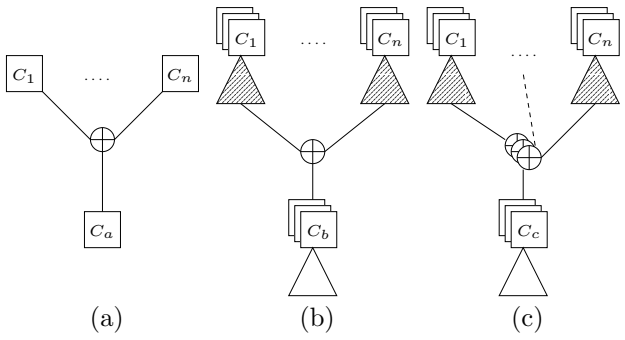

Fig. 1. Degrees of component variability

Figure 1 (b) and (c) show the case that all components have *configuration interfaces* in the form of feature diagrams [4] (the triangles). These feature diagrams express the components' variability. The stacked boxes indicate that a component can be instantiated to different variants. The shaded triangles indicate that C_b and C_c depend on specific *variants* of $C_1, ..., C_n$. Features that

remain to be selected by customers thus are local to the chosen top component (C_b resp. C_c).

The component dependencies of C_b are still fixed. For component C_c however, the component dependencies have become variable themselves: they depend on the selection of features described in the configuration interface of C_c. This allows components to be the units of variation. A consequence might be, for example, that when a client enables feature a, C_c requires component A. However, if feature b would have been enabled, C_c would depend on B. The set of constituent components of the resulting system may differ, according to the selected variant of C_c.

When composing C_a into a configuration suite, components $C_1, ..., C_n$ just have to be included. Components with variability, however, should be assembled into a suite guided by a valid selection of features declared by the top component (the component initially selected by the customer). Clients, both customers and requiring components, must select sets of features that are consistent with the feature diagram of the requested component.

How to establish these conditions automatically is deferred until after Sect. 4, where we introduce a domain specific language for describing components with variability.

4 Component Description Language

To formally evaluate component composition in the presence of variability, a language is needed to express the component variability described in Sect. 3. For this, Component Description Language (CDL) is presented. This language was designed primarily for the sake of exposition; the techniques presented here could just as well be used in the context of existing languages. The language will serve as a vehicle for the evaluation of the situation in Fig. 1 (c), that is: component dependencies may depend themselves on feature selections.

For the sake of illustration, we use the ATerm library as an example component. The ATerm library is a generic library for a tree like data structure, called Annotated Term (ATerm). It is used to represent (abstract) syntax trees in the ASF+SDF Meta-Environment [5], and it in many ways resembles the aforementioned `Tree` component. The library exists in both Java and C implementations. We have elicited some variable features from the Java implementation. The component description for the Java version is listed in Fig. 2.

A component description is identified by a name (`aterm-java`) and a version (`1.3.2`). Next to the identification part, CDL descriptions consist of two sections: the features section and the requires section.

The features section has a syntax similar to Feature Description Language (FDL) as introduced in [12]. FDL is used since it is easier to automatically manipulate than visual diagrams due to its textual nature. The features section contains definitions of composite features starting with uppercase letters. Composite features obtain their meaning from feature expressions that indicate how

sub-features are composed into composite features. Atomic features can not be decomposed and start with a lowercase letter.

The ATerm component exists in two implementations: a native one (implemented using the Java Native Interface, JNI), and a pure one (implemented in plain Java). The composite feature `Nature` makes this choice explicit to clients of this component. The feature obtains its meaning from the expression `one-of(native, pure)`. It indicates that either `native` or `pure` may be selected for the variable feature `Nature`, but not both. Both `native` and `pure` are atomic features. Other variable features of the ATerm-library are the use of maximal sub-term sharing (`Sharing`) and an inclusive choice of some export formats (`Export`). Additional constraints can be used to reduce the feature space. For example, the `sharedtext` feature enables the serialization of ATerms, so that ATerms can be written on file while retaining maximal sharing. Obviously, this feature requires the `sharing` feature. Therefore, the features section contains the constraint that `sharedtext` cannot be enabled without enabling `sharing`.

component description ⟨"aterm-java", "1.3.2"⟩
features
ATerm : **all**(Nature, Sharing, Export, visitors?)
Nature : **one-of**(native, pure)
Sharing : **one-of**(nosharing, sharing)
Export : **more-of**(sharedtext, text)
sharedtext **requires** sharing
requires
when sharing {
⟨"shared-objects", "1.3"⟩ **with** fasthash
}
when visitors {
⟨"JJTraveler", "0.4.2"⟩
}

Fig. 2. Description of `aterm-java`

The requires section contains component dependencies. A novel aspect of CDL is that these dependencies may be guarded by atomic features to state that they fire when a particular feature is enabled. These dependencies are *conditional* dependencies. They enable the specification of variable features for which components themselves are the unit of variation.

As an example, consider the conditional dependency on the `shared-objects` component which implements maximal sub-term sharing for tree-like objects. If the `sharing` feature is enabled, the ATerm component requires the `shared-objects` component. As a result, it will be included in the configuration suite. Note that elements of the requires section refer to *variants* of the required components. This means that component dependencies are configured in the same way as customers would configure a component. Configuration occurs by way of passing a list of atomic features to the required component. In the example this happens for the `shared-objects` dependency, where the variant containing optimized hash functions is chosen.

5 Guaranteeing Consistency

Since a configuration interface is formulated in FDL, we need a way to represent FDL feature descriptions in the SKB. Our prototype SKB is based on the calculus of binary relations, following [6]. The next paragraphs are therefore devoted to describing how feature diagrams can be translated to relations, and how querying can be applied to check configurations and obtain the required set of dependencies.

Transformation to Relations The first step proceeds through three intermediate steps. First of all, the feature definitions in the features section are inlined. This is achieved by replacing every reference to a composite feature with its definition, starting at the top of the diagram. For our example configuration interface, the result is the following feature expression:

```
all(one-of(native, pure), one-of(nosharing, sharing),
    more-of(sharedtext, text),visitors?)
```

The second transformation maps this feature expression and additional constraints to a logical proposition, by applying the following correspondences:

$$\begin{aligned}
\texttt{all}(f_1, ..., f_n) &\mapsto \textstyle\bigwedge_{i\in\{1,...,n\}} f_i \\
\texttt{more-of}(f_1, ..., f_n) &\mapsto \textstyle\bigvee_{i\in\{1,...,n\}} f_i \\
\texttt{one-of}(f_1, ..., f_n) &\mapsto \textstyle\bigvee_{i\in\{1,...,n\}} (f_i \wedge \neg(\bigvee_{j\in\{1,..,i-1,i+1,...,n\}} f_j))
\end{aligned}$$

Optional features reduce to $\top$. Atomic features are mapped to logical variables with the same name. Finally, a `requires` constraint is translated to an implication. By applying these mappings to the inlined feature expression, one obtains the following formula.

$$\begin{aligned}
&((native \wedge \neg pure) \vee (pure \wedge \neg native)) \wedge ((nosharing \wedge \neg sharing) \vee \\
&\quad (sharing \wedge \neg nosharing)) \wedge (sharedtext \vee text) \wedge (sharedtext \rightarrow sharing)
\end{aligned}$$

Checking the consistency of the feature diagram now amounts to obtaining *satisfiability* for this logical sentence. To achieve this, the formula is transformed to a Binary Decision Diagram (BDD) [1]. BDDs are logical expressions $\mathrm{ITE}(\varphi, \psi, \xi)$ representing if-then-else constructs. Using standard techniques from modelchecking any logical expression can be transformed into an expression consisting only of if-then-else constructs. If common subexpressions of this if-then-else expression are shared we obtain a directed acyclic graph which can easily be embedded in the relational paradigm. The BDD for the `aterm-java` component is depicted in Fig. 3.

Querying the SKB Now that we have described how feature diagrams are transformed to a form suitable for storing in the SKB, we turn our attention to the next step: the querying of the SKB for checking feature selections and obtaining valid configurations.

The BDD graph consists of nodes labeled by guards. Each node has two outgoing edges, corresponding to the boolean value a particular node obtains for a certain assignment. All paths from the root to $\top$ represent minimal assignments that satisfy the original formula.

A selection of atomic features corresponds to a partial truth-assignment. This assignment maps for each selected feature the corresponding guard to 1 (true). Let φ be the BDD derived from the feature diagram for which we want to check the consistency of the selection, then the meaning of a selection is defined as: $\{a_1, ..., a_n\} \mapsto \bigcup_{i \in \{1,...n\}} [a_i/1]$ when $a_i \in \varphi$. Checking whether this assignment can be part of a valuation amounts to finding a path in the BDD from the root to $\top$ containing the edges corresponding to the assignment. If there is no such path, the enabled features are incorrect. If there is such a path, but some other features must be enabled too, the result is the set of possible alternatives to extend the assignment to a valuation. The queries posted against the SKB use a special built-in query that generates all paths in a BDD. The resulting set of paths is then filtered according to the selection of features that has to be checked. The answer will be one of:

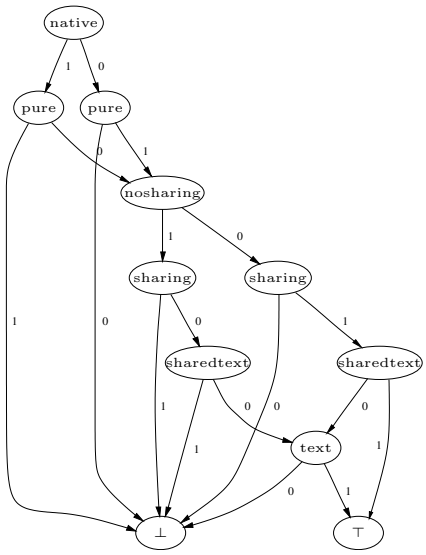

Fig. 3. BDD for `aterm-java`

- $\{\{f_1, ..., f_n\}, \{g_1, ..., g_m\}, ...\}$: a set of possible extensions of the selection, indicating an incomplete selection
- $\{\{\}\}$: one empty extension, indicating a correct selection
- $\{\}$: no possible extension, indicating incorrect selection

If the set of features was correct, the SKB is queried to obtain the set of configured dependencies that follow from the feature selection.

Take for example the selection of features {`pure`, `sharedtext`, `visitors`}. The associated assignment is $[pure/1][sharedtext/1]$. There is one path to $\top$ in the BDD that contains this assignment, so there is a valuation for this selection of features. Furthermore, it implies that the selection is not complete: part of the path is the truth assignment of `sharing`, so it has to be added to the set of selected features. Finally, as a consequence of the feature selection, both the JJTraveler and SharedObjects component must be included in the configuration suite.

6 Discussion

Related Work CDL is a domain specific language for expressing component level variability and dependencies. The language combines features previously

seen in isolation in other areas of research. These include: package based software development, module interconnection languages (MILs), and product configuration.

First of all, the work reported here can be seen as a continuation of package based software development [2]. In package based software development software is componentized in packages which have explicit dependencies and configuration interfaces. These configuration interfaces declare lists of options that can be passed to the build processes of the component. Composition of components is achieved through source tree composition. There is no support for packages themselves being units of variation. A component description in CDL can be interpreted as a package definition in which the configuration interface is replaced by a feature description. The link between feature models and source packages is further explored in [11]. However, variability is described external to component descriptions, on the level of the composition.

Secondly, CDL is a kind of module interconnection language (MIL). Although the management of variability has never been the center of attention in the context of MILs, CDL complies with two of the main concepts of MILs [7]:

- The ability to perform static type-checking at an intermodule level of description.
- The ability to control different versions and families of a system.

Static type-checking of CDL component compositions is achieved by model checking of FDL. Using dependencies and feature descriptions, CDL naturally allows control over different versions and families of a system. Variability in traditional MILs boils down to letting more than one module implement the same module interface. So modules are the primary unit of variation. In addition, CDL descriptions express variability without committing beforehand to a unit of variation.

We know of one other instance of applying BDDs to configuration problems. In [8] algorithms are presented to achieve interactive configuration. The configuration language consists of boolean sentences which have to be satisfied for configuration. The focus of the article is that customers can interactively configure products and get immediate feedback about their (valid or invalid) choices. Techniques from partial evaluation and binary decision diagrams are combined to obtain efficient configuration algorithms.

Contribution Our contribution is threefold. First, we have introduced variability at the component level to enable the product family approach in component-based product populations. We have characterized how component variability can be related to composition, and presented a formal language for the evaluation of this.

Secondly, we have demonstrated how feature descriptions can be transformed to BDDs, thereby proving the feasibility of a suggestion mentioned in the future work of [12]. Using BDDs there is no need to generate the exponentially large configuration space to check the consistency of feature descriptions and to verify user requirements.

Finally we have indicated how BDDs can be stored in a relational SKB which was our starting point for automated software delivery and generation of configurations.

The techniques presented in this paper have been implemented in a experimental relational expression evaluator, called RSCRIPT. Experiments revealed that checking feature selections through relational queries is perhaps not the most efficient method. Nevertheless, the representation of feature descriptions is now seamlessly integrated with the representation of other software facts.

Future Work Future work will primarily be geared towards validating the approach outlined in this paper. We will use the ASF+SDF Meta-Environment [5] as a case-study. The ASF+SDF Meta-Environment is a component-based environment to define syntax and semantics of (programming) languages. Although the Meta-Environment was originally targeted for the combination of ASF (Algebraic Specification Formalism) and SDF (Syntax Definition Formalism), directions are currently explored to parameterize the architecture in order to reuse the generic components (e.g., the user interface, parser generator, editor) for other specification formalisms [10]. Furthermore, the constituent components of the Meta-Environment are all released separately. Thus we could say that the development of the Meta-Environment is evolving from a component-based system towards a component-based product population. To manage the ensuing complexity of variability and dependency interaction we will use (a probably extended version of) CDL to describe each component and its variable dependencies.

In addition to the validation of CDL in practice, we will investigate whether we could extend CDL to make it more expressive. For example, in this paper we have assumed that component dependencies should be fully configured by their clients. A component client refers to a variant of the required component. One can imagine that it might be valuable to let component clients inherit the variability of their dependencies. The communication between client component and dependent component thus becomes two-way: clients restrict the variability of their dependencies, which in turn add variability to their clients. Developers are free to determine which choices customers can make, and which are made for them.

The fact that client components refer to variants of their dependencies induces a difference in binding time between user configuration and configuration during composition [3]. The difference could be made a parameter of CDL by tagging atomic features with a time attribute. Such a time attribute indicates the moment in the development and/or deployment process the feature is allowed to become active. Since all moments are ordered in a sequence, partial evaluation can be used to partially configure the configuration interfaces. Every step effects the binding of some variation points to variants, but may leave other features unbound. In this way one could, for example, discriminate features that should be bound by conditional compilation from features that are bound at activation time (e.g., via command-line options).

Acknowledgments Paul Klint and Gerco Ballintijn provided helpful comments on drafts of this paper. I thank Jurgen Vinju for carefully reading this article.

References

1. R. E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, Sept. 1992.
2. M. de Jonge. Package-based software development. In *Proc.: 29th Euromicro Conf.*, pages 76–85. IEEE Computer Society Press, 2003.
3. E. Dolstra, G. Florijn, and E. Visser. Timeline variability: The variability of binding time of variation points. In J. van Gurp and J. Bosch, editors, *Workshop on Software Variability Modeling (SVM'03)*, February 2003.
4. K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, SEI, CMU, Pittsburgh, PA, Nov. 1990.
5. P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, April 1993.
6. B. Meyer. The software knowledge base. In *Proc. of the 8th Intl. Conf. on Software Engineering*, pages 158–165. IEEE Computer Society Press, 1985.
7. R. Prieto-Diaz and J. M. Neighbors. Module interconnection languages. *The Journal of Systems and Software*, 6(4):307–334, November 1986.
8. M. H. Sørensen and J. P. Secher. From type inference to configuration. In *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*. Springer Verlag, 2002.
9. C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 2nd edition, 2002.
10. M. van den Brand, P. Moreau, and J. J. Vinju. Environments for Term Rewriting Engines for Free! In R. Nieuwenhuis, editor, *Proc. of the 14th International Conf. on Rewriting Techniques and Applications (RTA'03)*, volume 2706 of *LNCS*, pages 424–435. Springer-Verlag, 2003.
11. A. van Deursen, M. de Jonge, and T. Kuipers. Feature-based product line instantiation using source-level packages. In *Proceedings Second Software Product Line Conf. (SPLC2)*, Lecture Notes in Computer Science, pages 217–234. Springer-Verlag, 2002.
12. 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.
13. J. van Gurp, J. Bosch, and M. Svahnberg. On the Notion of Variability in Software Product Lines. In *Proceedings of the Working IEEE/IFIP Conf. on Software Architecture (WICSA'01)*, 2001.
14. R. van Ommering and J. 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.