Variability and Component Composition

Tijs van der Storm

REPORT SEN-R0403 FEBRUARY 25, 2004

ISSN 1386-369X

Variability and Component Composition

ABSTRACT
In component-based product populations, feature models have 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 component-level variability can be managed in the face of component composition. First, component variability and dependencies are described in a formal interface definition language. Secondly, these interfaces are checked for consistency by applying techniques from model checking. Thus, correct instantiation of product families by composition of components is guaranteed. The concepts and techniques presented here are the first step toward automated management of variability for web-based software delivery.

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; software delivery; configuration; feature diagrams; product family; composition; modelchecking; software knowledge base
Note: This work was sponsored in part by the dutch national research organization, NWO, Jacquard project Deliver.

Variability and Component Composition

Tijs van der Storm
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

ABSTRACT
In component-based product populations, feature models have 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 component-level variability can be managed in the face of component composition. First, component variability and dependencies are described in a formal interface definition language. Secondly, these interfaces are checked for consistency by applying techniques from model checking. Thus, correct instantiation of product families by composition of components is guaranteed. The concepts and techniques presented here are the first step toward automated management of variability for web-based software delivery.

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, software delivery, configuration, feature diagrams, product family, composition, modelchecking, software knowledge base

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

1. Introduction

Most of the time variability is considered at the level of one software product [vGBS01]. In a product family approach different variants of one product are derived from a set of core assets. However, in component-based product *populations* 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 describe how component interfaces can be enhanced with a description of component variability. To manage the ensuing complexity of configuration and component composition, we present techniques to formally verify the consistency of component interfaces, such that the conditions for correct component composition in the presence of component variability 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. We describe a novel kind of component interfaces that support the kind of variability we envision. 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. It provides a characterization of the relation between variability and composition.

Section 4 presents the domain specific language CDL for the description of component interfaces with support for component-level variability.

CDL will be the vehicle for the technical exposition of Sect. 5. The algorithm in that section implements the consistency requirements that were identified in Sect. 2.

Finally we provide some concluding remarks and our future work.

2. Towards Automated Management of Variability

2.1 Why Component Variability?

Software components are units of independent production, acquisition, and deployment [SGM02]. In a product family approach, many different variants of one system are derived by combining components in different ways. Components serve as building-blocks and units of variation for a family of software systems.

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 units of variation are not enough to realize all kinds of conceivable variability (e.g., crosscutting features).

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 enabled by inclusion of another component: the logging of debug information must be enabled via other mechanisms (e.g., conditional compilation).

The example clearly 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 natural place for the specification of this relation is in component interfaces. We use feature diagrams [EC00] for the description of component-level variability. They define a component's configuration interface. The required interface consists of versioned dependencies.

2.2 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 and delivery of software products and upgrades to numerous, possibly diverse customers. 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 configuration interfaces and feature selections. 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 precondition for correct composition of components is that required interfaces and configuration interfaces of two interacting components match. So if a component C_p requires $C_1, ..., C_n$, the required interfaces of C_p should be able to refer to the configuration interfaces of the dependent components $C_1, ..., C_n$. We will now asses what this means in the presence of component variability. Figure 1 depicts three possibilities for realizing component variability and composition.

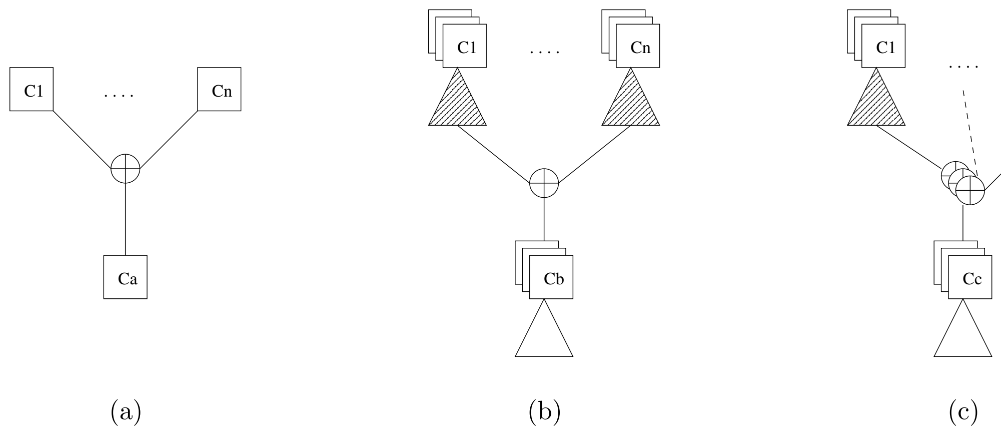

Figure 1: Degrees of component variability

The first case is when there is no variability at all. A component C_a requires components $C_1, ..., C_n$. Since we do not deal with required interface elements other than simple (versioned) dependencies, $C_1, ..., C_n$ should just be present somehow for the correct operation of C_p. The resulting system is the composition of C_p and $C_1, ..., C_n$, and all component dependencies that are transitively reachable from $C_1, ..., C_n$.

Figure 1 (b) and (c) show the case that all components have configuration interfaces in the form of feature diagrams (the triangles). These feature diagrams express the components' variability. The stacked boxes indicate that a component exists in many variants. The shaded triangles indicate that the required interfaces of C_b and C_c select the appropriate *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 "atoms of configuration" [Szy00]. 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 interfaced with feature diagrams, 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 describe 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 interfaces described in Sect. 3. For this, Component Description Language (CDL) is presented. 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. Note that the situation in Fig. 1 (c) subsumes the situation in Fig. 1 (b). This means that one is not forced to use components as units of variation, – conditional dependencies only provide this as a possibility.

For the sake of exposition, we use the ATerm library [vdBdJKO00] 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, 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.

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

Figure 2: Component description of the ATerm library in Java

A component description is identified by a name (`aterm-java`) and a version (`1.3.2`). This tuple serves as identifier for clients to address a component. Next to the identification part CDL descriptions consist of two sections: the configuration interface section and the required interface section.

The configuration interface section consists of a feature description in Feature Description Language (FDL) as introduced in [vDK02]. FDL is used since its textual nature allows easier automatic manipulation than visual diagrams. Clients, both customers and components, configure the configuration interface by selecting features.

The syntax of FDL consists of a diagram section and a constraints section. The diagram 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`).

The constraints section is used to reduce the feature space defined by the diagram section. It allows the expression of constraints on atomic features. These constraints can have the form of requirements and exclusions. 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 constraints section lists the rule that `sharedtext` cannot be enabled without enabling `sharing`.

Since components may depend on other components, the required interface contains versioned references to other components. A novel aspect of CDL is that these references 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.

The ATerm component has two conditional dependencies. One of the key features of the ATerm component is that the library implements maximal sub-term sharing, but its implementation is factored out in a separate component, which implements maximal sharing for any kind of object. So, 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.

Finally there is the optional `visitors` feature. Enabling this feature results in the inclusion of the JJTraveler component. JJTraveler is a Java component consisting of an implementation of the Visitor design pattern and a set of reusable visitor combinators [Vis01]. These visitor combinators can be used to define arbitrary traversals over ATerms in a generic fashion.

Recall that elements of the required interface 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

5.1 Plan

Because configuration interfaces are formulated in FDL, we need a model to represent FDL feature description in the SKB. Our prototype SKB is based on the calculus of binary relations, following [Mey85]. 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.

The top composite feature of a feature description is the entry point of configuration. Configuration is the selection of atomic features such that the diagram section and the constraint section are satisfied. Van Deursen and Klint [vDK02] showed that normalizing feature diagrams to a disjunctive normal form leads to an exponential blowup. Therefore, we use techniques from model-checking to avoid this in many cases.

Preparing the SKB for guaranteed correct composition in the presence of component variability is divided in two main steps.

Step 1 Transform the feature description to relations.

Step 2 Query these relations for a specific set of features.

A feature description is first translated to a boolean expression. The consistency of the feature description is obtained by checking satisfiability of this expression. The expression is then transformed to a binary-decision diagram (BDD) [Bry92]. Constructing a BDD for a formula is in itself a satisfiability test.

The BDD obtained in Step 1 is used to check feature selections against a feature description. First the BDD, which is a directed acyclic graph, is transformed to binary relations. These relations are stored in the SKB and can be queried for specific feature selections. The result is an indication of whether the selection of features was correct, incomplete or incorrect.

5.2 Details of Implementation

Transformation to Relations Step 1 proceeds through three intermediate steps. First of all, the diagram section of the feature description is 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:

$$\texttt{all}(f_1, ..., f_n) \mapsto \bigwedge_{i\in\{1,...,n\}} f_i \tag{5.1}$$

$$\texttt{more-of}(f_1, ..., f_n) \mapsto \bigvee_{i\in\{1,...,n\}} f_i \tag{5.2}$$

$$\texttt{one-of}(f_1, ..., f_n) \mapsto \bigoplus_{i\in\{1,...,n\}} f_i \tag{5.3}$$

$$f? \mapsto \top \tag{5.4}$$

Constraints are transformed as follows:

$$a_1 \texttt{ requires } a_2 \mapsto a_1 \to a_2 \tag{5.5}$$

$$a_1 \texttt{ excludes } a_2 \mapsto a_1 \to \neg a_2 \tag{5.6}$$

Atomic features are mapped to logical variables with the same name. In 5.3, the logical operator $\oplus$ is defined as:

$$\bigoplus_{i\in\{1,...,n\}} f_i = \bigvee_{i\in\{1,...,n\}} (f_i \wedge \neg(\bigvee_{j\in\{1,..,i-1,i+1,...,n\}} f_j)) \tag{5.7}$$

By applying these mappings to the inlined feature expression, one obtains the following formula.

$$\begin{aligned}&((native \wedge \neg pure) \vee (pure \wedge \neg native))\wedge\\ &((nosharing \wedge \neg sharing) \vee (sharing \wedge \neg nosharing))\wedge\\ &\qquad (sharedtext \vee text) \wedge (sharedtext \to sharing)\end{aligned} \tag{5.8}$$

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 BDD. BDDs are logical formulas that obey the following syntax:

$$\Phi ::= \top \mid \bot \mid \text{ITE}(g, \Phi, \Phi) \tag{5.9}$$

Where g is logical variable that acts as guard. The $\text{ITE}(\varphi, \psi, \xi)$ is called is an if-then-else construct; it is equivalent to $(\varphi \wedge \psi) \vee (\neg\varphi \wedge \xi)$.

The efficiency of BDDs depends on a chosen total ordering on guards, i.e. atomic features. We have chosen to define this ordering according to the order of occurrence of atomic features during a top-down, left-to-right traversal of the diagram section. The Shannon expansion converts any logical formula to an ordered BDD using the following fact: for every logical variable g, occurring in φ, the next equation holds:

$$\varphi = \text{ITE}(g, \varphi[g/\top], \varphi[g/\bot]) \tag{5.10}$$

Read from left to right, the equation is a rewrite rule used in the process of transforming a logical formula to a BDD. Our algorithm repeatedly applies this rule for each guard, in the order of the total ordering on guards, while at the same time simplifying the formulas. For example, $\varphi \wedge \top$ is reduced to φ. Upon termination we are left with a binary decision tree consisting solely of if-then-else expressions, with $\top$ and $\bot$ as leaves. The final step consists of sharing common subexpressions, transforming the tree to 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.

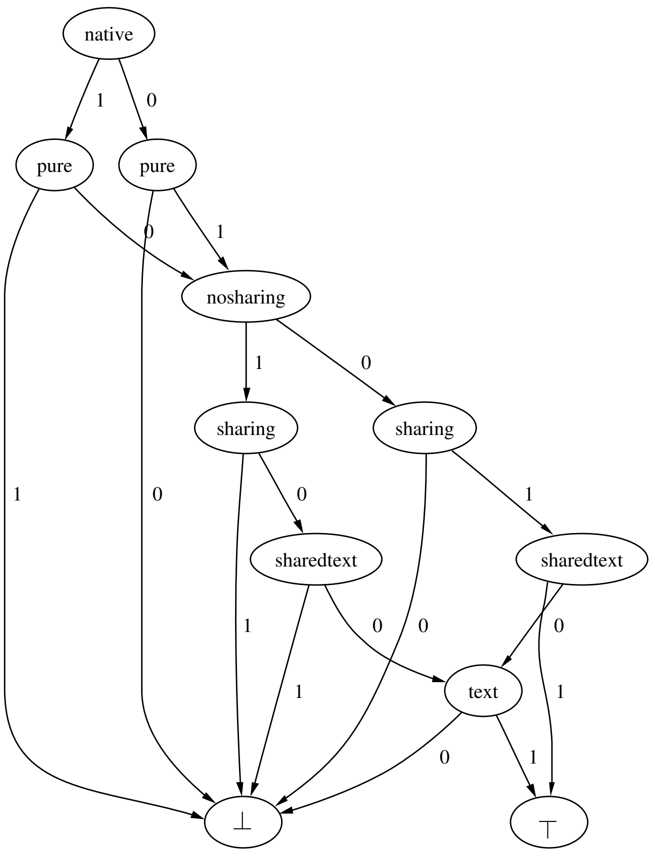

Figure 3: BDD for feature diagram of `aterm-java`

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 step 2: the querying of the SKB for checking feature selections and obtaining valid configurations. These queries are expressed in the calculus of (nested) binary relations.

The BDD graph consists of nodes labeled by guards. Each node has two exiting 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. Recall that the transformation of feature descriptions to BDDs itself is a consistency check: any formula that is a contradiction reduces to the unique BDD $\bot$. So if the result is other than $\bot$ the feature diagram is consistent (i.e. satisfiable).

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 the selection is defined as:

$$\{a_1, ..., a_n\} \quad \mapsto \quad \bigcup_{i \in \{1, ... n\}} \quad [a_i/1] \quad \text{when } a_i \in \varphi \tag{5.11}$$

Checking whether this assignment can be part of a valuation amounts to finding a path in the BDD (viewed as a graph) 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:

- $\{\{f_1, ..., f_n\}, \{g_1, ..., g_m\}, ...\}$: 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 $\{\texttt{pure}, \texttt{sharedtext}, \texttt{visitors}\}$. 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. Concluding Remarks

6.1 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 base software development [dJ03]. 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 [dJ02]. 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 [vDdJK02]. However, variability is described external to component descriptions.

Viewed from a level of finer granularity, 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 [PDN86]:

- 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 versioned 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 [SS02] 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.

6.2 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 is 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 [vDK02]. 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 configuration artifacts.

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 relational SKB does indeed smoothly integrate this functionality with relational knowledge representation.

6.3 Future Work

Future work will primarily be geared towards validating the approach outlined in this paper. We will use the ASF+SDF Meta-Environment [Kli93, vdBvDH^{+}01] for this 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 [vdBMV03]. 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 [DFV03, DFdJV03]. 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 specific features unbound. In this way, for example, one could 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

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

[DFdJV03] Eelco Dolstra, Gert Florijn, Merijn de Jonge, and Eelco Visser. Capturing timeline variability with transparent configuration environments. In Jan Bosch and Peter Knauber, editors, *IEEE Workshop on Software Variability Management (SVM'03)*, Portland, Oregon, USA, May 2003. IEEE.

[DFV03] Eelco Dolstra, Gert Florijn, and Eelco Visser. Timeline variability: The variability of binding time of variation points. In Jilles van Gurp and Jan Bosch, editors, *Workshop on Software Variability Modeling (SVM'03)*, number 2003-7-01 in IWI preprints, Groningen, The Netherlands, February 2003. Reseach Institute of Computer Science and Mathematics, University of Groningen.

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

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

[EC00] Ulrich W. Eisenecker and Krzysztof Czarnecki. *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley, 2000.

[Kli93] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, April 1993.

[Mey85] B. Meyer. The software knowledge base. In *Proceedings of the 8th International Conference on Software Engineering*, pages 158–165. IEEE Computer Society Press, 1985.

[PDN86] Ruben Prieto-Diaz and James M. Neighbors. Module interconnection languages. *The Journal of Systems and Software*, 6(4):307–334, November 1986.

[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.

[SS02] Morten Heine Sørensen and Jens Peter Secher. From type inference to configuration. In *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated* *to Neil D. Jones*. Springer Verlag, 2002.

[Szy00] Clemens Szyperski. Component software and the way ahead. In G. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*. Cambridge University Press, UK, 2000.

[vdBdJKO00] M.G.J. van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient annotated terms. *Software, Practice and Experience*, 30(3):259–291, 2000.

[vdBMV03] M.G.J. van den Brand, P.E. Moreau, and J. J. Vinju. Environments for Term Rewriting Engines for Free! In R. Nieuwenhuis, editor, *Proceedings of the 14th International Conference on Rewriting Techniques and Applications (RTA'03)*, volume 2706 of *LNCS*, pages 424–435. Springer-Verlag, 2003.

[vdBvDH$^+$01] M. G. J. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In R. Wilhelm, editor, *Compiler Construction (CC '01)*, volume 2027 of *Lecture Notes in Computer Science*, pages 365–370. Springer-Verlag, 2001.

[vDdJK02] A. van Deursen, M. de Jonge, and T. Kuipers. Feature-based product line instantiation using source-level packages. In *Proceedings Second Software Product Line Conference (SPLC2)*, Lecture Notes in Computer Science, pages 217–234. Springer-Verlag, 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.

[vGBS01] Jilles van Gurp, Jan Bosch, and Mikael Svahnberg. On the Notion of Variability in Software Product Lines. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, 2001.

[Vis01] Joost Visser. Visitor combination and traversal control. *ACM SIGPLAN Notices*, 36(11):270–282, November 2001. OOPSLA 2001 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications.