Generic Feature-Based Software Composition

Tijs van der Storm

Centrum voor Wiskunde en Informatica
P.O. Box 94079, 1090 GB Amsterdam
The Netherlands
storm@cwi.nl

Abstract. Bridging problem domain and solution in product line engineering is a time-consuming and error-prone process. Since both domains are structured differently (features vs. artifacts), there is no natural way to map one to the other. Using an explicit and formal mapping creates opportunities for consistency checking and automation. This way both the configuration and the composition of product instances can be more robust, support more product variants and be performed more often.

1 Introduction

In product line engineering, automatic configuration of product line instances still remains a challenge [1]. Product configuration consists of selecting the required features and subsequently instantiating a software product from a set of implementation artifacts. Because features capture elements of the problem domain, automatic product composition requires the explicit mapping of features to elements of the solution domain. From a feature model we can then generate tool support to drive the configuration process.

However, successful configuration requires consistent specifications. For instance, a feature specification can be inconsistent if selecting one feature would require another feature that excludes the feature itself. Because of the possibly exponential size of the configuration space, maintaining consistency manually is no option.

We investigate how to bridge the "white-board distance" between problem space and solution space [15] by combining both domains in a single formalism based on feature descriptions [20]. White-board distance pertains to the different levels of abstraction in describing problem domain on the one hand, and solution domain on the other hand. In this paper, feature descriptions are used to formally describe the configuration space in terms of the problem domain. The solution domain is modeled by a dependency graph between artifacts.

By mapping features to one or more solution space artifacts, configurations resulting from the configuration task map to compositions in the solution domain. Thus it becomes possible to derive a configuration user interface from the feature model to automatically instantiate valid product line variants.

1.1 Problem-Solution Space Impedance Mismatch

The motivation for feature-based software composition is based on the following observations: solution space artifacts are unsuitable candidates for reasoning about the configurability in a product line. Configuration in terms of the problem domain, however, must stand in a meaningful relation to those very artifacts if it should be generally useful. Let's discuss each observation in turn.

First, if software artifacts can be composed or configured in different ways to produce different product variants it is often desirable to have a high-level view on which compositions are actually meaningful product instances. That is, the *configuration space* should be described at a high level of abstraction. If such configuration spaces are expressed in terms of problem space concepts, it is easier to choose which variant a particular consumer of the software actually needs. Finally, such a model should preferably be a formal model in order to prevent inconsistencies and configuration mistakes.

The second observation concerns the value of relating the configuration model to the solution space. The mental gap between problem space and solution space complicates keeping the configuration model consistent with the artifacts. Every time one or more artifacts change, the configuration model may become invalid. Synchronizing both realms without any form of tool support is a time-consuming and error-prone process. In addition, even if the configuration model is used to guide the configuration task, there is the possibility of inconsistencies in both the models and their interplay.

From these observations follows that in order to reduce the effort of configuring product lines and subsequently instantiating product variants tool support is needed that helps detecting inconsistencies and automates the manual, error-prone task of collecting the artifacts for every configuration. This leads to the requirements for realizing automatic software composition based on features.

- The configuration interface should be specified in a language that allows *formal* consistency checking. If a configuration interface is consistent then this means there are valid configurations. Only valid configurations must be used to instantiate products. Such configurations can be mapped to elements of the solution domain.
- A model is needed that relates features to artifacts in the solution space, so that if a certain feature is selected, all relevant artifacts are collected in the final product. Such a mapping should respect the (semantic) relations that exist between the artifacts. For the mapping to be as applicable as possible no assumptions should be made about programming language or software development methodology.

1.2 Related Work

This work is directly inspired by the technique proposed in [9]. In that position paper feature diagrams are compared to grammars, and parsing is used to check the consistency of feature diagrams. Features are mapped to software

packages. Based on the selection of features and the dependencies between packages, the product variant is derived. Our approach generalizes this technique on two accounts: first we allow arbitrary constraints between features, and not only structural ones that can be verified by parsing. Second, in our approach *combinations* of features are mapped to artifacts, allowing more control over which artifact is required when.

There is related work on feature oriented programming that provides features with a direct solution space semantics. For instance, in AHEAD [2] features form elements in an algebra that can be synthesized into software components. Although this leaves open the choice of programming language it assumes that it is class-based. Czarnecki describes a method of mapping features to model elements in an model driven architecture (MDA) setting [7]. By "superimposing" *all* variants on top of UML models, a product can be instantiated by selectively disabling variation points.

An even more fine grained approach is presented in [17] where features become first-class citizens of the programming language. Finally, a direct mapping of features to a component role model is described in [12].

These approaches all, one way or the other, merge the problem domain and the solution domain in a single software development paradigm. In our approach we keep both domains separate and instead relate them through an explicit modeling step. Thus our approach does not enforce any programming language, methodology or architecture beforehand, but instead focuses on the possibility of automatic configuration and flexibility.

Checking feature diagrams for consistency is an active area of research [20, 6, 16] but the level of formality varies. The problem is that checking the consistency is equivalent to propositional satisfiability, and therefore it is often practically infeasible. Our approach is based on BDDs [19], a proven technique from model checking, which often makes the exponential configuration space practically manageable.

1.3 Contributions

The contributions of this paper can be summarized as follows:

- Using an example we analyze the challenges of bridging the gap between problem space and solution space. We identify the requirements for the explicit and controlled mapping of features to software artifacts.
- We propose a formal model that allows both worlds to be bridged in order to achieve (solution space) composition based on (problem space) configuration. Instances of the model are checked for consistency using scalable techniques widely used in model-checking.
- The model is unique in that it does not dictate programming language, is independent of software development methodology or architectural style, and does not require up-front design. The latter in turn allows the approach to be adopted late in the development process or in the context of legacy software.

Organization of this paper In the following section, Sect. 2, feature diagrams [13] are introduced as a model for the configuration space of product lines. Feature diagrams are commonly used to elicit commonality and variability of software systems during domain analysis [21]. They can be formally analyzed so they are a viable option for the first requirement.

Next, in Sect. 2.3 we present an abstract model of the solution space. Because we aim for a generic solution, this model is extremely simple: it is based on the generic notion of *dependency*. Thus, the solution space is modeled by a dependency graph between artifacts. Artifacts include any kind of file that shapes the final software product. This includes source files, build files, property files, locale files etc.

Then, in Sect. 3 we discuss how feature diagrams and dependency graphs should be related in order to allow automatic composition. The formalization of feature diagrams is described in Sect. 3.2, thus enabling the application of model-checking techniques for the detection of inconsistencies. How both models are combined is described in Sect. 4. This combined model is then used to derive product instances. Finally we present some conclusions and provide directions for future work.

2 Problem and Solution Space Models

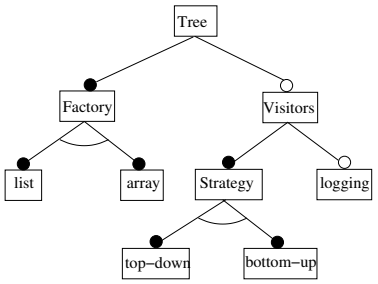


Fig. 1. Problem space of a small example visualized as a feature diagram

2.1 Introduction

To be able to reason about the interaction between problem space an solution space, models are required that accurately represent the domains in a sufficiently formal way. In this section we introduce feature diagrams as a model for the problem space, and dependency graphs for the solution space.

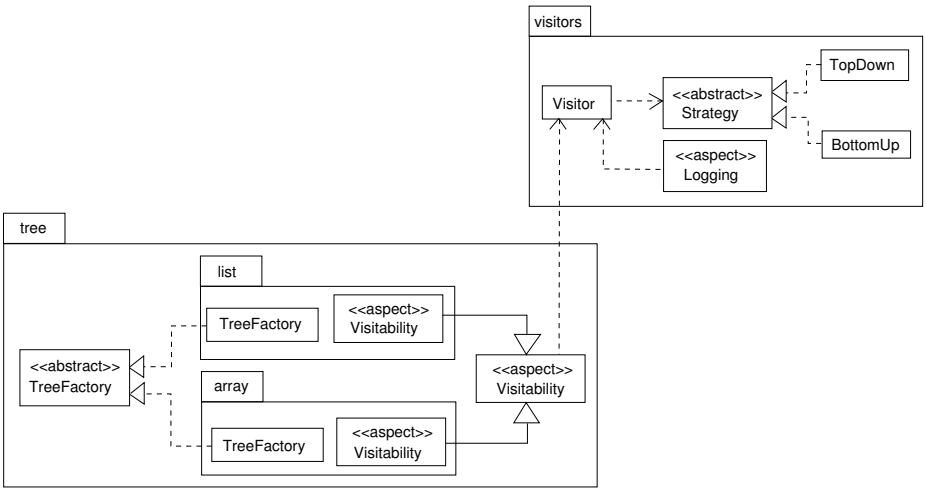


Fig. 2. UML view of an example product line

2.2 Problem Space: Feature Diagrams

Figure 1 shows a graphical model of a small example's problem space using feature diagrams [13]. Feature diagrams have been used to elicit commonality and variability in domain engineering. A feature diagram can be seen as a specification of the configuration space of a product line.

In this example, the top feature, Tree, represents the application, in this case a small application for transforming tree-structured documents, such as parse trees. The Tree feature is further divided in two sub features: Factory and Visitors. The Visitors feature is optional (indicated by the open bullet), but if it is chosen, a choice must be made between the top-down or bottom-up alternatives of the Strategy feature and optionally there is the choice of enabling logging support when traversing trees. Finally, the left sub-feature of Tree, named Factory, captures a mandatory choice between two, mutually exclusive, implementations of trees: one based on lists and the other based on arrays.

Often these diagrams are extended with arbitrary constraints between features. For instance one could state that the array feature *requires* the logging feature. Such constraints make visually reasoning about the consistency of feature selection with respect to a feature diagram much harder. In order to automate such reasoning a semantics is needed. Many approaches exist, see e.g. [16, 3, 4]. In earlier work we interpreted the configuration problem as satisfiability problem and we will use that approach here too [19]. The description consistency checking of feature diagrams is deferred to Sect. 3.2.

2.3 Solution Space: Implementation Artifacts

The implementation of the example application consists of a number of Java classes and AspectJ files [14]. Figure 2 shows a tentative design in UML. The implementation of the transformation product line is divided over two components: a tree component and visitors component. Within the tree component the Abstract Factory design pattern is employed to facilitate the choice among list- and array-based trees. In addition to the choice between different implementations, trees can optionally be enhanced with a `Visitable` interface by weaving an aspect. This enables that clients of the tree component are able to traverse the trees by using the visitors component. So weaving in the Visitability aspect causes a dependency on the visitors component.

2.4 Artifact Dependency Graphs

What is a suitable model of the solution space? In this paper we take a an abstract stance and model the solution space by a directed acyclic dependency graph. In a dependency graph nodes represent artifacts and the edges represent dependencies between them. These dependencies may be specified explicitly or induced by the semantics of the source. As an example of the latter: a Java class file has a dependency on the class file of its superclass. Another example are aspects that depend on the classes they will be weaved in. For the example the dependencies are shown in Fig. 3. The figure shows dependencies of three kinds: subtype dependency (e.g. between list.Tree and Tree), aspect dependency (between Visitability and Tree), collaboration dependency (between Visitor and Strategy).

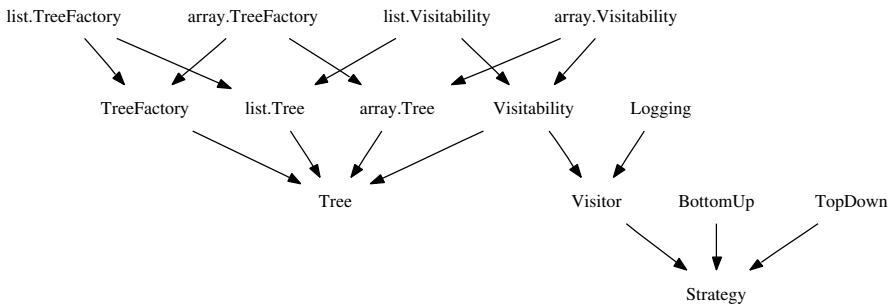


Fig. 3. Solution space model of the example: dependency graph between artifacts

Dependency graphs are consistent, provided that the dependency relation conforms to the semantics of the artifacts involved and provided that every node in the graph has a corresponding artifact. A set of artifacts is consistent

with respect to a dependency graph if it is closed under the dependency relation induced by that graph.

A nice property of these graphs is that, in theory, every node in it represents a valid product variant (albeit a useless one most of the time). If we, for instance, take the Visitability node as an example, then we could release this 'product' by composing every artifact reachable from the Visitability node. So, similar to the problem space of the previous section, the solution space is also a kind of configuration space. It concisely captures the possibilities of delivery.

3 Mapping Features to Artifacts

3.1 Introduction

Now that the problem space is modeled by a feature diagram and the solution space by a dependency graph how can we bridge the gap between them? Intuitively one can map each feature of the feature diagram to one or more artifacts in the dependency graph. Such an approach is graphically depicted in Fig. 4.

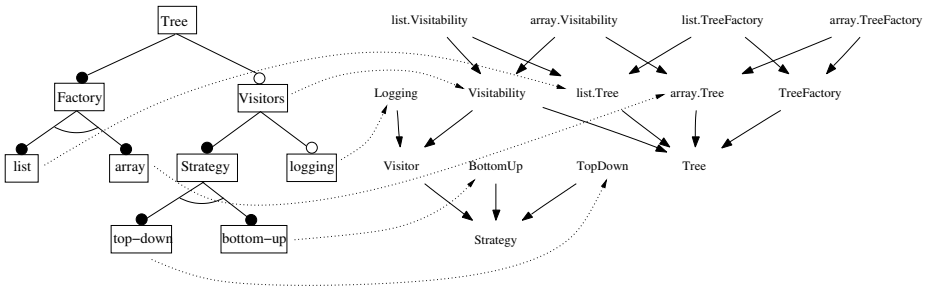


Fig. 4. Partial mapping of features to artifacts

The figure shows the feature diagram together with the dependency graph of the previous section. Arrows from features to the artifacts indicate which artifact should be included if a feature is selected. For instance, if the top-down strategy is chosen to visit trees, then the TopDown implementation will be delivered together with all its dependencies (i.e. the Strategy interface). Note that the feature mapping is incomplete: selecting the Visitors feature includes the Visitability aspect, but it is unspecified which concrete implementation (list.Visitability or array.Visitability) should be used. The graphical depiction thus is too weak to express the fact that if *both* array/list and Visitors is chosen, both the array.Visitability/list.Visitability and Visitability artifacts are required. In Sect. 4 this problem will be addressed by expressing mapping as constraints between features and artifacts.

Tree : **all**(Factory, Visitors**?**)
Factory : **one-of**(list, array)
Visitors : **all**(Strategy, logging**?**)
Strategy : **one-of**(top-down, bottom-up)

Fig. 5. Textual FDL feature description of the example

3.2 Feature Diagram Semantics

Features	Logic
feature	boolean formula
atomic and composite features	atoms
configurability	satisfiability
configuration	valuation
validity of a configuration	satisfaction

Table 1. Feature descriptions as boolean formulas

This section describes how feature diagrams can be checked for consistency. We take a logic based approach that exploits the correspondence between feature diagrams and propositional logic (see Table 1). Since graphical formalisms are less practical for building tool support, we use a textual version of feature diagrams, called Feature Description Language (FDL) [20]. The textual analog of feature diagram in Fig. 1 is displayed in Fig. 5. Composite features start with an upper-case letter whereas atomic feature start in lower-case. Composing features is specified using connectives, such as, **all** (mandatory), **one-of** (alternative), **?** (optional), and **more-of** (non-exclusive choice). In addition to representing the feature diagram, FDL allows arbitrary constraints between features.

For instance, in the example one could declare the constraint "array **requires** logging". This constraint has the straightforward meaning that selecting the array feature should involve selecting the logging feature. Because of these and other kinds of constraints a formal semantics of feature diagrams is needed, because constraints may introduce inconsistencies not visible in the diagram, and they may cause the invalidity of certain configurations, which is also not easily discerned in the diagram.

3.3 Configuration Consistency

The primary consistency requirement is internal consistency of the feature description. An inconsistent feature description cannot be configured, and thus it would not be possible to instantiate the corresponding product. An example of an inconsistent feature description would be the following:

A : **all**(b, c)
b **excludes** c

Feature b excludes feature c, but they are defined to be mandatory for A. This is a contradiction if `A` represents the product. Using the correspondence between feature descriptions and boolean formulas (cf. Table 1), we can check the consistency of a description by solving the satisfiability problem of the corresponding formula.

Configuration spaces of larger product lines quickly grow to exponential size. It is therefore essential that scalable techniques are employed for the verification and validation of feature descriptions and feature selections respectively. Elsewhere, we have described a method to check the logical consistency requirements of component-based feature diagrams [19]. That technique is based on translating component descriptions to logical formulas called binary decision diagrams (BDDs) [5]. BDDs are logical if-then-else expressions in which common subexpressions are shared; they are frequently used in model-checking applications because they often represent large search spaces in a feasible way. Any propositional formula can be translated to a BDD. A BDD that is different from falsum ($\bot$) means that the formula is satisfiable.

A slightly different mapping is used here to obtain the satisfiability result. The boolean formula derived from the example feature description is as follows:

$$(Tree \rightarrow Factory) \wedge$$
$$(Factory \rightarrow ((list \ \wedge \ \neg array) \ \vee \ (\neg list \ \wedge \ array))) \wedge$$
$$(Visitors \rightarrow Strategy) \wedge$$
$$(Strategy \rightarrow ((top\text{-}down \ \wedge \ \neg bottom\text{-}up) \ \vee \ (\neg top\text{-}down \ \wedge \ bottom\text{-}up))))$$

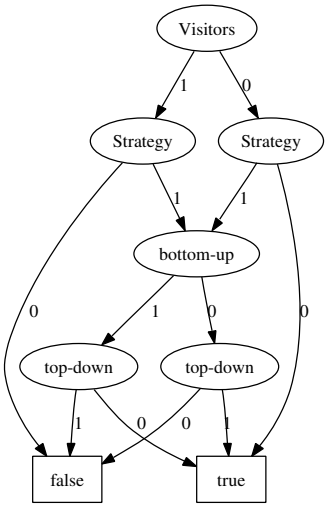
Note how all feature names become logical atoms in the translation. Feature definitions of the form $Name : Expression$ become implications, just like "requires" constraints. The translation of the connectives is straightforward. Such a boolean formula can be converted to a BDD using standard techniques (see for instance [11] for an elegant approach).

The resulting BDD can be displayed as a directed graph where each node represents an atom and has two outcoming edges corresponding to the two branches of the if-then-else expression. Figure 6 shows the BDD for the Visitors feature bot as a graph and if-then-else expression. As one can see from the paths in the graph, selecting the Visitors feature means enabling the Strategy feature. This in turn induces a choice between the top-down and bottom-up features. Note that the optional logging feature is absent from the BDD because it is not constrained by any of the other variables.

4 Selection and Composition of Artifacts

4.1 Introduction

If a feature description is found to be consistent, it can be used to generate a configuration user interface. Using this user interface, an application engineer would




```
if Visitors then
  if Strategy then
    if bottom-up then
      if top-down then ⊥ else ⊤ fi
    else
      if top-down then ⊤ else ⊥fi
    fi
  else
    ⊥
  fi
else
  if Strategy then
    if bottom-up then
      if top-down then ⊥ else ⊤ fi
    else
      if top-down then ⊤ else ⊥fi
    fi
  else
    ⊤
  fi
fi
```

Fig. 6. BDD for the Visitors feature

select features declared in the feature description. Selections are then checked for validity using the BDD. The selection of features, called the configuration, is then used to instantiate the product. Sets of selected features correspond to a sets of artifacts. Let's call these the (configuration) induced artifacts. The induced artifacts form the initial composition of the product. Then, every artifact that is reachable from any of the induced artifacts in the dependency graph, is added to the composition.

4.2 Configuration and Selection

In Sect. 3 we indicated that mapping single features to sets of artifacts was not strong enough to respect certain constraints among the artifacts. The example was that the concrete Visitability aspects (array.Visitability and list.Visitability) were not selected if the Visitors feature were only mapped to the abstract aspect Visitability. To account for this problem we extend the logical framework introduced in Sect. 3.2 with constraints between features and artifacts. Thus, mappings become requires constraints (implications) that allow us to include artifacts when certain *combinations* of features are selected. The complete mapping of the example would then be specified as displayed in Fig. 7.

The constraints in the figure – basically a conjunction of implications – are added to the feature description. Using the process described in the previous section, this hybrid 'feature' description is translated to a BDD. The set of required artifacts can then be found by partially evaluating the BDD with the selection of features. This results in a, possibly partial, truth-assignment for the atoms representing artifacts. Any artifact atom that gets assigned $\top$ will be included in the composition together with the artifacts reachable from it in

list **and** Visitors **requires** *list.Visitability*
array **and** Visitors **requires** *array.Visitability*
list **requires** *list.TreeFactory*
array **requires** *array.TreeFactory*
top-down **requires** *TopDown*
bottom-up **requires** *BottomUp*
logging **requires** *Logging*

Fig. 7. Mapping features to artifacts

the dependency graph. Every artifact that gets assigned $\perp$ will not be included. Finally, any artifact that did not get an implied assignment may or may not be included, but at least is not required by the selection of features.

Figure 8 shows all possible configurations for the example product line. The configurations are shown as a nested tree map. Every box represents a valid sub composition induced by the feature at left-hand side, upper corner. The artifacts contained in each composition are shown in italics. The figure shows that even this very small product line already exposes 12 product variants.

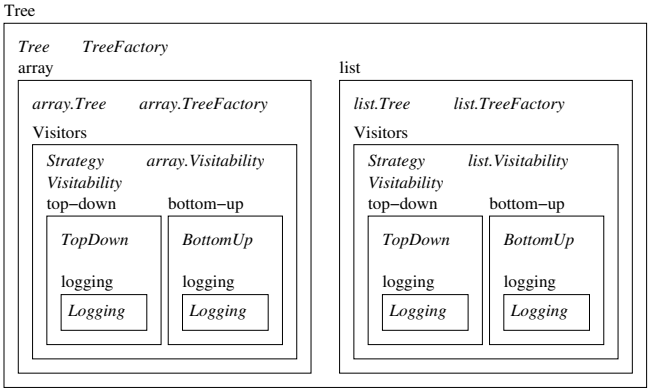


Fig. 8. All configurations/compositions of the example as a nested tree-map

4.3 Composition Methods

In the previous subsection we described how the combination of problem space feature models can be linked to solution space dependency graphs. For every valid configuration of the feature description we can derive the artifacts that should be included in the final composition. However, how to enact the composition was left unspecified. Here we discuss several options for composing the artifacts according to the dependency graph.

In the case of the example composing the Java source files entails collecting them in a directory an compiling the source files using `javac` and AspectJ. However, this presumes that the artifacts are actually Java source files, which may be a too fine granularity. Next we describe three approaches to composition that support different levels of granularity:

- Source tree Composition [8]
- Generation of a build scripts [22]
- Container-based dependency injection [10]

Source Tree Composition Source tree composition is based on *source packages*. Source packages contain source code and have an abstract build interface. Each source package explicitly declares which other packages it requires during build, deployment and/or operation. The source trees contained in these packages can be composed to obtain a composite package. This package has a build interface that is used to build the composition by building every sub-packages in the right order with the right configuration parameters.

Applying this to our configuration approach this would mean that artifacts would correspond to source packages. Every valid selection of features would map to a set of root packages. From these root packages every transitively required packages can be found and subsequently be composed into a composite package, ready for distribution.

Build Script Generation An approach taken in the KOALA framework [22] is similar to source tree composition but works at the level of C-files. In KOALA a distinction is made between *requires* interfaces (specifying dependencies of a component) and *provides* interfaces (declaring the function that a component has to offer). The composition algorithm of KOALA takes these interfaces and the component definitions (describing part-of hierarchies) and subsequently generates a `Makefile` that specifies how a particular composition should be built.

Again, this could be naturally applied in our context of dependency graphs. The artifacts would be represented by the interfaces and the providing components. The dependency graph then follows from the requires interfaces.

Dependency Injection Another approach to creating the composition based on feature selections would consist of generating configuration files (or configuration code) for a dependency injection container implementation [10]. Dependency injection is a object-oriented design principle that states that every class should only reference interfaces in its code. Concrete implementations of these interfaces are then "injected" into a class via the constructor of via setter methods. How component classes are connected together ("wiring") is specified separately from the components.

In the case of Java components, we could easily derive the dependencies of those clasess by looking at the interface parameters of their constructors and setters. Moreover, we can statically derive which classes implement those interfaces (which also induces a dependency). Features would then be linked to

these implementation classes. Based on the dependencies between the interfaces and classes one could then generate the wiring code.

5 Conclusions

5.1 Discussion: Maintaining the Mapping

Since problem space and solution space are structured differently, bridging the two may induce a high maintenance penalty if changes in either of the two invalidate the mapping. It is therefore important that the mapping of feature to artifacts is explicit, but not tangled.

The mapping of features to artifacts presented in this paper allows the automatic derivation of product instances based on dependency graphs, but the mapping itself must be maintained by hand. Maintaining the dependency relation manually is no option since it continually co-evolves with the code base itself, but often these relations can be derived from artifacts automatically (e.g., by static analysis).

It is precisely the separation of feature models and dependency graphs makes maintaining the mapping manageable if the dependency graphs are available automatically. For certain nodes in the graph we can compute the transitive closure, yielding all artifacts transitively required from the initial set of nodes. This means that a feature has to be mapped only to the *essential* (root) artifact; all other artifacts follow from the dependency graph.

Additionally, changes in the dependencies between artifacts (as follows from the code base) have less severe consequences on such mappings. On other words, the coevolution between feature model and mapping on the one hand, and the code base on the other is much less severe. This reduces the cost of keeping problem space and solution space in sync.

5.2 Conclusion & Future Work

The relation between problem space and solution space in the presence of variability poses both conceptual and technical challenges. We have shown that both worlds can be brought together by importing solution space artifacts into the domain of feature descriptions. By modeling the relations among software artifacts explicitly and interpreting the mapping of combinations of features to artifacts as constraints on the hybrid configuration space, we obtain a coherent formalism that can be used for generating configuration user interfaces. On the technical level we have proposed the use BDDs to make automatic consistency checking of feature descriptions and mapping feasible in practice. Configurations are input to the composition process which takes into account the complex dependencies between software artifacts.

This work, however, is by no means finished. The formal model, as discussed in this paper, is still immature and needs to be investigated in more detail. More analyses could be useful. For instance, one would like to know which configurations a certain artifact participates in order to better assess the impact of certain

modifications to the code-base. Another direction we will explore is the implementation of a feature evolution environement that would help in maintaining feature models and their relation to the solution space.

A case-study must be performed to see how the approach would work in practice. This would involve building a tool set that allows the interactive editing, checking and testing of feature descriptions, which are subsequently fed into a product configurator, similar to the CML2 tool used for the Linux kernel [18]. The Linux kernel itself would provide a suitable case to test our approach.

References

1. Don Batory, David Benavides, and Antonio Ruiz-Cortés. Automated analyses of feature models: Challenges ahead. *Communications of the ACM*, December 2006. To appear.
2. Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. In *Proceedings of the 25th International Conf. on Software Engineering (ICSE-03)*, pages 187–197, Piscataway, NJ, May 3–10 2003. IEEE Computer Society.
3. David Benavides, Pablo Trinidad Martín-Arroyo, and Antonio Ruiz Cortés. Automated reasoning on feature models. In Oscar Pastor and João Falcão e Cunha, editors, *Advanced Information Systems Engineering, 17th International Conference, CAiSE 2005, Porto, Portugal, June 13-17, 2005, Proceedings*, volume 3520 of *Lecture Notes in Computer Science*, pages 491–503. Springer, 2005.
4. Yves Bontemps, Patrick Heymans, Pierre-Yves Schobbens, and Jean-Christophe Trigaux. Semantics of feature diagrams. In Tomi Männistö and Jan Bosch, editors, *Proc. of Workshop on Software Variability Management for Product Derivation (Towards Tool Support)*, Boston, August 2004.
5. Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.
6. 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.
7. Krzysztof Czarnecki and Michal Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In Robert Glück and Michael R. Lowry, editors, *Generative Programming and Component Engineering, 4th International Conference, GPCE 2005, Tallinn, Estonia, September 29 - October 1, 2005, Proceedings*, volume 3676 of *Lecture Notes in Computer Science*, pages 422–437. Springer, 2005.
8. 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.
9. M. de Jonge and J. Visser. Grammars as feature diagrams. draft, April 2002.
10. Martin Fowler. Inversion of control containers and the dependency injection pattern. Online: `http://www.martinfowler.com/articles/injection.html`, February 2006.
11. J. F. Groote and J. C. van de Pol. Equational binary decision diagrams. Technical Report SEN-R0006, Centre for Mathematics and Computer Science (CWI), Amsterdam, 2000.

12. Anton Jansen. Feature based composition. Master's thesis, Rijksuniversiteit Groning, 2002.
13. 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, November 1990.
14. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353. Springer-Verlag, 2001.
15. Paul Klint and Tijs van der Storm. Reflections on feature-oriented software engineering. In Christa Schwanninger, editor, *Workshop on Managing Variabilities Consistently in Design and Code held at the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004)*, 2004. Available from: `http://www.cwi.nl/~storm`.
16. 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.
17. Christian Prehofer. Feature-oriented programming: A fresh look at objects. In *ECOOP*, pages 419–443, 1997.
18. Eric S. Raymond. The CML2 language. In *9th International Python Conference*, 2001. Available at: `http://www.catb.org/~esr/cml2/cml2-paper.html`. (accessed October 2006).
19. 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.
20. 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.
21. Jilles van Gurp, Jan Bosch, and Mikael Svahnberg. On the Notion of Variability in Software Product Lines. In *Proceedings of the Working IEEE/IFIP Conf. on Software Architecture (WICSA'01)*, 2001.
22. 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.