Language-Independent Reference Checking in Software Product Lines

Sven Apel, Wolfgang Scholz, and Christian Lengauer
University of Passau, Germany
{apel, scholz, lengauer}@fim.uni-passau.de

Christian Kästner
Philipps University Marburg, Germany
kaestner@informatik.uni-marburg.de

ABSTRACT

Feature-Oriented Software Development (FOSD) is a paradigm for the development of software product lines. A challenge in FOSD is to guarantee that all software systems of a software product line are correct. Recent work on type checking product lines can provide a guarantee of type correctness without generating all possible systems. We generalize previous results by abstracting from the specifics of particular programming languages. In a first attempt, we present a reference-checking algorithm that performs key tasks of product-line type checking independently of the target programming language. Experiments with two sample product lines written in Java and C are encouraging and give us confidence that this approach is promising.

Categories and Subject Descriptors: D.3.3 [**Software**]: Programming Languages—*Formal Definitions and Theory*; D.3.3 [**Software**]: Programming Languages—*Language Constructs and Features*

General Terms: Languages, Reliability, Design

Keywords: Feature-Oriented Software Development, Software Product Lines, Type Systems, FeatureHouse, FeatureTweezer

1. INTRODUCTION

Feature-Oriented Software Development (FOSD) is a paradigm for the development of software product lines [3, 11, 13, 26]. The key idea is to modularize software systems in terms of features. A *feature* is a unit of functionality of a software system that satisfies a requirement, represents a design decision, or provides a configuration option [3]. Typically, with a set of features, a developer describes the commonalities and variabilities of a family of software systems of a particular domain (i.e., a software product line).

There are various ways of making the features of a software product line explicit in its document and code base [22]. Compositional approaches such as AHEAD [11] encapsulate the code that belongs to a feature in a cohesive and composable unit. Once we have made features explicit in terms of cohesive and composable units, a customized software system can be generated automatically, simply by selecting a valid subset of features. Commonly, a feature model defines which feature selections are valid [13], and tools are available that check selections for validity [12, 23].

A challenge addressed in recent work is to guarantee that *every* valid feature selection produces a type-correct program [5, 14, 15, 20, 28]. The problem is that, during the life time of a software product line, the feature model and the features' implementations tend to diverge. That is, programs may be well-typed that are not valid in terms of the feature model and programs may be ill-typed that are valid in terms of the feature model. The latter case is problematic because such errors are usually detected only when the program in question is generated based on a feature selection. Due to the possibly very large number of different valid feature selections, this may happen late in the development process, leading to high costs and ripple effects. Generating and compiling all programs is feasible only for small product lines. Product-line type checking addresses this problem by checking the entire code base of a product line once against the product line's feature model to ensure that no valid feature selection produces an erroneous program.

Contemporary product-line–checking approaches are tailored to specific programming languages, development tools, or formalisms, which limits principally their general applicability (see Sec. 5). We would like to explore how far we can implement product-line checking independently of the particular language or formalism used. This approach is inspired by previous work on language-independent feature composition tools [6, 7] and type systems [1]. Naturally, there is a trade-off between generality and expressiveness. That is, in our quest to increase generality, we may lose expressiveness, but we argue that this path is worth exploring and that our initial results are encouraging.

We concentrate on a subclass of possible type errors: *dangling references*. In a product line, code of one feature may refer to code of another feature (e.g., in the form of a method invocation or field access); if the former feature is selected and the latter is not, the former has a dangling reference, reported by the type system. We propose a language-independent model of programs with references, define (two

FOSD'10 October 10, 2010, Eindhoven, The Netherlands

variants of) an algorithm for detecting potential dangling references, and apply our prototypical implementation to two sample product lines written in Java and C.

2. PRELIMINARIES

Our language-independent model of feature-oriented product lines is based on previous work on feature algebra [8] and feature composition tools [6]. For illustration, we use the small example of a variable graph data structure, which is motivated by one of our case studies in Section 4. The graph example consists of the three features GRAPH, WEIGHTS, and DIJKSTRA whose meanings will become clear later. It has been implemented in FeatureHouse, a tool that supports the composition of code written in a number of languages, among them Java [6].

2.1 Feature Structure Trees

We describe the structure of a feature, independently of the programming language, by a *feature structure tree* (FST). An FST organizes the feature's structural elements (e.g., files, classes, fields, or methods) hierarchically. Figure 1 depicts an excerpt of the Java implementation of feature GRAPH and its representation in the form of an FST. One can think of an FST as a stripped-down abstract syntax tree that contains only essential information. The nature of this information depends on the degree of granularity at which software artifacts are to be used, analyzed, or composed, as we discuss below.

```
1  package graph;
2  class Graph {
3    List<Node> nodes;
4    List<Edge> edges;
5    void add(Node n) {
6      ...
7    }
8    ...
9  }
10 class Node {
11   int id; ...
12 }
13 class Edge {
14   Node src, dest; ...
15 }
```

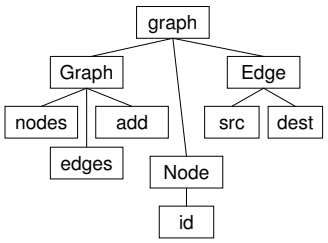

Figure 1: Implementation and FST of feature GRAPH (excerpt).

For example, the FSTs we use to represent Java code contain nodes that represent packages, classes, fields, methods, etc. They do not contain information on the internal structure of methods and so on. A different level of granularity would represent only packages and classes but not methods or fields as FST nodes (coarser granularity). Yet another level of granularity would represent additionally statements or expressions as nodes (finer granularity) [21].

Each node of an FST is labeled with a name and a type (types are not shown in Figure 1, for brevity). We call two nodes *compatible* if they have the same name and type and compatible parents. A node's name[1] corresponds to the name of the structural element it represents and a node's type corresponds to the syntactic category to which the element belongs. For example, class Graph is represented by node Graph of type class (type omitted in Figure 1, for brevity). We must consider both the name and the type to prevent ambiguities.

Note that, at the granularity we chose for Java, the order of nodes could be arbitrary, but this may be different at a finer granularity (e.g., the order of statements matters) and it may also differ for other languages (e.g., the order of C functions and of overlapping Haskell patterns matters) [6]. The FST model has been used to formalize and implement feature composition based on FST superimposition, of which we report elsewhere [6, 8].

It has been shown that the FST model is very general. It can be used with different languages including object-oriented (e.g., Java and C#), imperative (e.g., C), functional (e.g., Haskell), modeling (e.g., UML), and relational languages (e.g., Alloy) [2, 4, 6, 9]. Next, we extend the FST model with references.

2.2 Adding References

First, we introduce the concept of a reference into the FST model. Then, we discuss key design decisions.

References. A product line consists of multiple features, each of which is represented by an FST. Commonly, there are dependences between the individual features. One feature may extend or use another (e.g., in that it invokes a method belonging to the other feature). This kind of reference is common in many languages. Examples of references are field accesses in Java, references between grammar product rules in JavaCC, associations in UML, and so on. Hence, we extend the FST model by references. A reference is a pair of a name of source FST element and a name of a destination FST element. The two elements need not to point to the same FST.

Let us illustrate references by means of our graph example. In Figure 2, we show the implementation and FST of feature WEIGHTS. It refines class Edge of feature GRAPH by adding a new field weight.[2]

```
1 package graph;
2 class Edge {
3   int weight;
4   ...
5 }
```

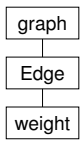

Figure 2: Implementation and FST of feature WEIGHTS (excerpt).

In Figure 3, we show the implementation and FST of feature DIJKSTRA, which implements Dijkstra's algorithm for solving the shortest-path problem. It introduces class Dijkstra that, at some point, accesses field weight of class Edge.

The three features refer to one another: WEIGHTS refers to GRAPH and DIJKSTRA refers to GRAPH and WEIGHTS. A reference has a source (left-hand side) and a target (right-hand side). The source consists of the feature's name and the element's fully-qualified name. The target consists only of the element's fully-qualified name. In the graph example, we

[1] Depending on the language and purpose, a name could be a simple identifier, a signature, etc.

[2] When composing feature GRAPH and feature WEIGHTS, the two declarations of class Edge are merged; this is a form of mixin composition [11].

```
1 package graph;
2 class Dijkstra {
3   Node[] shortestPath(Node n) {
4     ... int w = edge.weight; ...
5   }
6 }
```

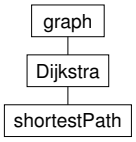

Figure 3: Implementation and FST of feature DIJKSTRA **(excerpt).**

have the following references, excluding the inner references of a feature to itself:

(WEIGHTS, `Edge`) → (`Edge`)
(DIJKSTRA, `Dijkstra.shortestPath`) → (`Node`)
(DIJKSTRA, `Dijkstra.shortestPath`) → (`Edge`)
(DIJKSTRA, `Dijkstra.shortestPath`) → (`Edge.weight`)
...

It is important to note that the target element can be part of different features, so it is not fixed to which element another element points—references are resolved after the desired features have been selected by a user to generate a final program. For example, there may be multiple features that introduce different kinds of weights. It is the task of the reference checker to ensure that there is a proper target for each reference in every valid feature selection.

Discussion. Programs and documents written in various languages can be represented by FSTs [6] and references are a language-independent concept. In fact, the extended FST model represents (a subset of) the context-sensitive abstract grammar of a language, whereas the plain FST model without references represents (a subset of) the context-free abstract grammar. Essentially, the first design decision was to detach the reference model from the underlying language and to base it entirely on FSTs. This way, we attain language independence but may reduce expressiveness in that we cannot represent the full type structure of a language, which may interfere with reference checking. Language independence also implies that references (and FSTs) have to be represented in a general format. For a product line to be checked, FSTs and references have to be extracted by (language-specific) code analysis tools. Then, reference checking is generic and uniform.

A second design decision we made is that we model references as *pairs* of FST elements. This is the simplest model possible and we use it until we encounter the need for a more complex model. In some languages, we may need references with multiple possible targets. Furthermore, some languages may need a more dynamic view of references, for instance, to take dynamic binding into account.

The third design decision we made is that a reference contains, beside the source and target elements, also the source feature; in contrast, the target feature is not contained in the reference and undefined until the user selects a set of features. The rationale is that, if a piece of code contains a reference, we know to which feature the piece belongs, but we do not know which feature provides a proper target for the reference—in fact, there may be multiple features. However, in some languages, there may be situations in which the type of the target element is relevant (e.g., a field with type String instead of int). Since we currently do not support full typing, we cannot address this issue without losing language independence.

FSTs and their references provide insight into the structural interactions between features. The knowledge about the features of a product line, their references, and their valid selections (i.e., the feature model) allows us to formulate a language-independent reference-checking algorithm.

2.3 Feature Models

Before we describe the reference-checking algorithm, we repeat briefly the basics of feature models. A *feature model* describes the valid feature selections of a software product line [19]. There are different approaches and notations for describing feature models [12]. We use the approach of Batory in which a feature model is represented by a propositional formula [10]. The formula contains, for each feature, a boolean variable and expresses the constraints between features. Most other notations can be translated to propositional formulas.

A propositional formula describing the variability of our graph example could look as follows:

$$(\text{WEIGHTS} \vee \text{DIJKSTRA}) \Rightarrow \text{GRAPH} \qquad (1)$$

The formula states that, whenever WEIGHTS or DIJKSTRA are selected, then also GRAPH has to be selected. A consequence is that the features WEIGHTS and DIJKSTRA are optional and independent (which can lead to a dangling reference, as we will explain shortly).

Solver technology can be employed to answer a number of questions on feature models including whether a feature selection is valid or whether a feature is always, sometimes, or never present when another feature is present [12], which is interesting information for reference checking. For example, feature GRAPH is *always* present when one of the other two features is present and feature WEIGHTS is *sometimes* present when feature DIJKSTRA is present.

Typically, a feature model describes the variability of a product line without considering a particular implementation. Hence, the variability of a domain does not necessarily need to be consistent with the variability of the corresponding implementation. There may be valid feature selections that lead to programs with type errors. For example, assuming the feature model of Equation 1, there is a valid feature selection that leads to an incorrect program in our graph example. By selecting GRAPH and DIJKSTRA only, we get a type error because DIJKSTRA refers to field weight, which belongs to the non-selected feature WEIGHTS.

The graph example illustrates that implementation variability may differ from domain variability. Both kinds of variability can be described by feature models. The feature model that describes the implementation variability of our graph example, henceforth called the *implementation model*, is:

$$(\text{WEIGHTS} \vee \text{DIJKSTRA}) \Rightarrow \text{GRAPH} \quad \wedge \\ (\text{DIJKSTRA} \Rightarrow \text{WEIGHTS}) \qquad (2)$$

Compared to the feature model of Equation 1, henceforth called the *domain model*, it contains an additional constraint that states that, whenever DIJKSTRA is selected, WEIGHTS has to be selected, too. A key task of reference checking is to derive information on implementation variability and to check it against domain variability to discover potential dangling references.

Algorithm 1 Global reference-checking algorithm.

Require: $\mathbb{F}$:= set of feature names
Require: FT := FST table
Require: RT := reference table
Require: DM := domain feature model
1: *// derive formula of implementation feature model*
2: $IM := \bigwedge_{((f,src),tgt)\in RT}(f \Rightarrow \bigvee_{f_i\in\mathbb{F},tgt\in FT(f_i)} f_i)$
3: *// check whether the two models are consistent*
4: $sln := \mathsf{solve}(DM \Rightarrow IM)$
5: **if** $sln \neq true$ **then**
6: *// determine counterexample*
7: $cex := \mathsf{counter}(DM \Rightarrow IM)$
8: $\mathsf{print}(cex)$
9: *// identify dangling references*
10: $dref := \{(ref=((f,src),tgt)) \mid$
11: $ref \in RT, f \in cex, (\nexists f \in cex : tgt \in FT(f))\}$
12: **for all** $(ref = ((f,src),tgt)) \in dref$ **do**
13: *// identify features with potential targets*
14: $ptgt := \{f \mid f \in (\mathbb{F}\backslash cex),\ tgt \in FT(f)\}$
15: $\mathsf{print}(ref)$
16: $\mathsf{print}(ptgt)$
17: **end for**
18: **end if**

Algorithm 2 Local reference-checking algorithm.

Require: $\mathbb{F}$:= set of feature names
Require: FT := FST table
Require: RT := reference table
Require: DM := domain feature model
1: **for all** $(ref=((f,src),tgt)) \in RT$ **do**
2: *// derive formula of reference*
3: $RM := (f \Rightarrow \bigvee_{f_i\in\mathbb{F},tgt\in FT(f_i)} f_i)$
4: *// check for consistency*
5: $sln := \mathsf{solve}(DM \Rightarrow RM)$
6: **if** $sln \neq true$ **then**
7: *// determine counterexample*
8: $cex := \mathsf{counter}(DM \Rightarrow RM)$
9: $\mathsf{print}(cex)$
10: *// identify features with potential targets*
11: $ptgt := \{f \mid f \in (\mathbb{F}\backslash cex),\ tgt \in FT(f)\}$
12: $\mathsf{print}(ref)$
13: $\mathsf{print}(ptgt)$
14: **end if**
15: **end for**

3. REFERENCE-CHECKING ALGORITHM

There are two variants of our reference-checking algorithm, each of which has been inspired by a different branch of previous work (see Sec. 5): the *global reference-checking algorithm* creates a single propositional formula (i.e., the implementation feature model) that covers all references [14, 15, 28]; the *local reference-checking algorithm* creates a propositional formula for each reference that covers exactly the constraints implied by this reference [5, 20].

Global Reference-Checking Algorithm. In Algorithm 1, we list the global variant of the reference-checking algorithm including comments. It takes as input the following ingredients of a product line: the domain feature model, the FSTs of all features, and the references between FST elements. Based on this information, the global variant of the algorithm proceeds in three steps:

1. The input FSTs and references are analyzed to obtain the implementation feature model. To this end, for each reference, all proper target elements are searched. At least one target (i.e., its feature) must be present when the source feature is present. This requirement is added as a disjunctive clause to the propositional formula of the implementation model (Line 2).
2. A SAT solver checks whether the implementation model is consistent with the domain model (Line 4).
3. If the two models are not consistent, a counterexample is generated, which is a set of features that contains dangling references (Line 7). Based on the counterexample, for each dangling reference, all features with proper targets are identified (Lines 12–17).

Local Reference-Checking Algorithm. In Algorithm 2, we list the local variant of the reference-checking algorithm, including comments. Like in the global variant, it takes as input the domain feature model, the FSTs of all features, and the references between FST elements. The algorithm proceeds in three steps:

1. Rather than creating a single propositional formula for the entire implementation model, one propositional formula per reference is generated, which describes the constraints implied by the reference, called the *reference model*. Again, at least one target of the reference (i.e., its feature) must be present when the source feature is present (Line 3).
2. A SAT solver checks whether the constraints imposed by each single reference are consistent with the domain model (Line 5).
3. If the constraints of some reference are not satisfied, a counterexample is generated (Line 8). Based on the counterexample, for the dangling reference in question, all features with proper targets are identified (Line 11).

```
1 erroneous feature selection:
2   [Graph,Dijkstra]
3 dangling reference:
4   (Dijkstra.shortestPath, Edge.weight)
5   in feature 'Dijkstra'
6 features that provide proper targets:
7   [Weights]
```

Figure 4: Output of FeatureTweezer when checking the graph example.

Discussion. Both variants of the reference-checking algorithm have a similar input-output behavior. They expect domain and structural information and provide information on dangling references and potential target features. For our graph example, both variants of the algorithm would produce an output like the one shown in Figure 4.

So what is the difference between the two variants and why have researchers invented them in the first place? A key difference is the size and number of propositional formulas to be checked for consistency with the domain model. In the global variant, we have a single, possibly large formula; in the local variant, we have many smaller formulas. This difference may be crucial for performance, an issue that has

gained too little attention in the past. Admittedly, there is initial evidence that extracting a complete implementation model is possible in linear time [27] and that solving large formulas that represent feature models is possible in practice in polynomial time [25]. But there is also evidence that solving many small formulas is efficient because intermediate results can be cached and reused [5]. In the future, we intend to address this issue systematically. A contribution of our language-independent model (and tool) is that we can represent both variants at an abstract level, reveal their principal differences, and provide a basis for experiments.

Another difference between the local and global variant of the reference-checking algorithm is error reporting. The local variant is finer-grained in that it identifies potential dangling references directly and points to the corresponding locations in the code. The global variant searches first for an erroneous feature selection and identifies then potential dangling references, but only for this selection. The local variant identifies all potential dangling references. This makes the debugging process less iterative and more efficient.

4. PROTOTYPE AND CASE STUDIES

As a proof of concept, we have been developing a prototype of a product-line reference checker in Haskell, called FeatureTweezer.[3] Checking for dangling references in a product line, FeatureTweezer expects the product line's FSTs, the references, and the feature model. Currently, an FST is encoded as a set of prefix-closed identifiers (the prefix encodes the path in the FST), each of which denotes an FST element. A reference is encoded as a pair of FST element identifiers. A feature model is encoded in the GUIDSL format [10], but, as illustrated in Figure 5, alternative formats are possible. We depict the input data for our graph example to FeatureTweezer in Figure 6, simplified and adapted for presentation purposes.

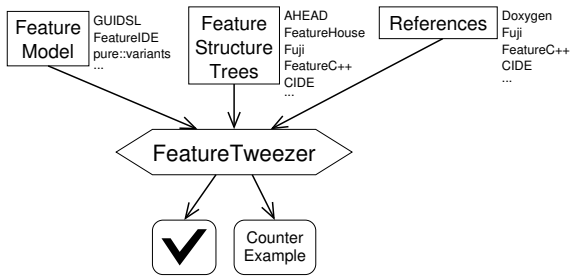

Figure 5: Input and output of FeatureTweezer.

FeatureTweezer processes all input information, transforms it into a suitable internal format, and feeds it into the reference-checking algorithm (see Sec. 3). To check reference constraints and implementation models against the domain model, we use *funsat*[4], an open-source native Haskell SAT solver. It provides a counterexample if the models are not consistent. This information is used to find sources of dangling references and potential features that provide proper targets.

To gather experience with FeatureTweezer, we applied it to two sample product lines: the graph product line (GPL) of Lopez-Herrejon and Batory [24] and the feature-oriented email client of Hall [18]. The former product line is implemented in Java and the latter is implemented in C. A key challenge is to extract the necessary information for FeatureTweezer (FSTs, references, and feature model). For the purpose of our initial experiments, we chose a pragmatic approach. We extracted the FSTs with FeatureHouse[5] and the references with Doxygen[6] and CCVisu[7]. We extended FeatureHouse to export the FSTs of a product line to the expected format and we configured Doxygen and CCVisu such that the element identifiers in the references correspond to the element identifiers in the FSTs; the feature models were available as part of the sample product lines.

The fact that we were able to check two product lines written in two different languages illustrates the potential of our approach. We did not find bugs in the two sample product lines. This is not surprising because they are rather small and well-tested. However, the product lines were useful for testing our tool by introducing errors artificially. The size of the sample product lines do not allow us to draw any conclusions on the differences in performance and resource consumption between the local and the global reference-checking algorithm. In further work, we intend to check more and larger product lines, written in different languages, to discover real bugs and to measure performance and resource consumption.

domain feature model

```
1 GraphExample : Graph [Weights] [Dijkstra] ;
```

feature structure trees (type feature element)

```
1 ClassDecl Graph Graph
2 FieldDecl Graph Graph.nodes
3 FieldDecl Graph Graph.edges
4 ...
5 ClassDecl Weights Edge
6 FieldDecl Weights Edge.weight
7 ...
8 ClassDecl Dijkstra Dijkstra
9 MethodDecl Dijkstra shortestPath
```

references (type (feature element) element)

```
1 ClassRef (Weight Edge) Edge
2 ClassRef (Dijkstra Dijkstra.shortestPath) Node
3 ClassRef (Dijkstra Dijkstra.shortestPath) Edge
4 MethodRef (Dijkstra Dijkstra.shortestPath) Edge.weight
5 ...
```

Figure 6: Input data of the graph example for FeatureTweezer (simplified).

5. RELATED WORK

Our work on reference checking feature-oriented product lines has been motivated by previous work on type systems for product lines [5, 14, 15, 20, 28]. Reference checking is an important subset of type checking, so we believe we have taken an important step toward a general tool suite for product-line engineering.

Thaker et al. developed a type system for feature-oriented product lines, based on Java, that does not check all in-

[3] FeatureTweezer, including the examples and case studies, is available on the Web: `http://www.fosd.de/FT/`

[4] `http://github.com/dbueno/funsat/`

[5] `http://www.fosd.de/fh/`

[6] `http://www.doxygen.org`

[7] `http://www.sosy-lab.org/~dbeyer/CCVisu/`

dividual programs but the individual feature implementations [28]. In a number of case studies, they found numerous hidden errors using their type rules. Our global variant of the reference-checking algorithm is inspired by their type-checking algorithm, which generates a single, large propositional formula representing implementation variability. Furthermore, Delaware et al. developed a formal model of the type system of Thaker et al. and proved its soundness [15].

Even previously to the work of Thaker et al., Czarnecki and Pietroszek presented an automatic verification procedure for ensuring that no ill-formed UML model template instances will be generated from a valid feature selection [14], which also uses a global reference-checking algorithm. That is, they type check product lines that consist not of Java programs but of annotated UML models (they use OCL constraints to express and implement a kind of type system for UML; annotations denote features).

Our local reference-checking algorithm is inspired by our own work on formal type systems for product lines [5, 20]. Kästner and Apel have developed the formal calculus CFJ based on a subset of Java and a set of type rules for checking annotation-based product lines [20]. Like in the approach of Czarnecki and Pietroszek, and in contrast to AHEAD, variability is implemented with #ifdef-like directives or similar annotations on the source code [21]. Kästner and Apel use a local type-checking algorithm to guarantee type correctness. Similarly, Apel et al. [5] define a formal model of a feature-oriented language and a corresponding product-line type system based on an compositional approach and a subset of Java.

All of the approaches we discussed so far are tailored to specific languages and core languages. An interesting aspect is that our model is able to express reference checking in both composition-based and annotation-based product lines. Both can be reduced to FSTs and references.

Tartler et al. demonstrate that implementation models of C code including preprocessor directives can be extracted in linear time [27]. As in the global reference-checking algorithm, they extract the implementation model once in the form of a single propositional formula. Mendonca et al. demonstrate that consistency checking in the global approach is possible in practice in polynomial time [25]. Apel et al. illustrate how caching can be used to scale the local variant of reference checking by reusing intermediate results [5]. These pieces of work illustrate that there is a potential for tuning the performance of type checking product lines. Our approach and tool can provide a basis for further experiments in this direction.

6. CONCLUSION

We presented a language-independent reference checking algorithm for feature-oriented product lines. To this end, we extended the existing model of feature structure trees with references. Our algorithm checks an entire product line and reports whether any valid feature selection results in a program that contains a dangling reference. The algorithm is based on FSTs, extended with references, and the feature model of a product line. We developed a prototypical tool called FeatureTweezer, which we used to check two sample product lines written in Java and C for dangling references. We believe that our work is a first step toward a more general understanding of and more general tools for feature-oriented product lines implemented using both compositional units and annotations.

In further work, we plan to extend our approach based on language-independent, cross-language, or extensible type systems [1,16,17] with other well-formedness criteria such as mutual exclusion, typing, and subtyping. Furthermore, we plan to extend existing feature algebraic models with references and to reason about the effects of references on algebraic properties. Finally, we intend to conduct further case studies of different domains, written in different languages and provide evidence on the generality of our approach, to discover real bugs, and to measure performance and resource consumption of local and global reference checking.

Acknowledgments

Wolfgang Scholz is supported by the German Research Foundation (DFG—AP 206/2-1). Kästner's work is supported by the European Research Council (ERC #203099).

7. REFERENCES

[1] S. Apel and D. Hutchins. A Calculus for Uniform Feature Composition. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(5):Article 19, 2010.

[2] S. Apel, F. Janda, S. Trujillo, and C. Kästner. Model Superimposition in Software Product Lines. In *Proceedings of the International Conference on Model Transformation (ICMT)*, volume 5563 of *LNCS*, pages 4–19. Springer-Verlag, 2009.

[3] S. Apel and C. Kästner. An Overview of Feature-Oriented Software Development. *Journal of Object Technology (JOT)*, 8(5):49–84, 2009.

[4] S. Apel, C. Kästner, A. Größlinger, and C. Lengauer. Feature (De)composition in Functional Programming. In *Proceedings of the International Conference on Software Composition (SoftComp)*, volume 5634 of *LNCS*, pages 9–26. Springer-Verlag, 2009.

[5] S. Apel, C. Kästner, A. Größlinger, and C. Lengauer. Type Safety for Feature-Oriented Product Lines. *Automated Software Engineering – An International Journal (2010)*, 17(3):251–300, 2010.

[6] S. Apel, C. Kästner, and C. Lengauer. FeatureHouse: Language-Independent, Automated Software Composition. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 221–231. IEEE CS, 2009.

[7] S. Apel and C. Lengauer. Superimposition: A Language-Independent Approach to Software Composition. In *Proceedings of the International Symposium on Software Composition (SoftComp)*, volume 4954 of *LNCS*, pages 20–35. Springer-Verlag, 2008.

[8] S. Apel, C. Lengauer, B. Möller, and C. Kästner. An Algebraic Foundation for Automatic Feature-Based Program Synthesis. *Science of Computer Programming (SCP)*, 75(11):1022–1047, 2010.

[9] S. Apel, W. Scholz, C. Lengauer, and C. Kästner. Detecting Dependences and Interactions in Feature-Oriented Design. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*. IEEE CS, 2010.

[10] D. Batory. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the*

International Software Product Line Conference (SPLC), volume 3714 of *LNCS*, pages 7–20. Springer-Verlag, 2005.

[11] D. Batory, J. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering (TSE)*, 30(6):355–371, 2004.

[12] D. Benavides, S. Segura, and A. Ruiz-Cortes. Automated Analysis of Feature Models 20 years Later: A Literature Review. *Information Systems*, 35(6):615–636, 2010.

[13] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[14] K. Czarnecki and K. Pietroszek. Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 211–220. ACM Press, 2006.

[15] B. Delaware, W. Cook, and D. Batory. Fitting the Pieces Together: A Machine-Checked Model of Safe Composition. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, pages 243–252. ACM Press, 2009.

[16] M. Grechanik, D. Batory, and D. Dewayne. Design of Large-Scale Polylingual Systems. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 357–366. IEEE CS, 2004.

[17] N. Haldiman, M. Denker, and O. Nierstrasz. Practical, Pluggable Types for a Dynamic Language. *Computer Languages, Systems and Structures*, 35(1):48–62, 2009.

[18] R. Hall. Fundamental Nonmodularity in Electronic Mail. *Automated Software Engineering*, 12(1):41–79, 2005.

[19] 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, 1990.

[20] C. Kästner and S. Apel. Type-Checking Software Product Lines – A Formal Approach. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 258–267. IEEE CS, 2008.

[21] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 311–320. ACM Press, 2008.

[22] C. Kästner, S. Apel, and M. Kuhlemann. A Model of Refactoring Physically and Virtually Separated Features. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 157–166. ACM Press, 2009.

[23] C. Kästner, T. Thüm, G. Saake, J. Feigenspan, T. Leich, F. Wielgorz, and S. Apel. FeatureIDE: Tool Framework for Feature-Oriented Software Development. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 611–614. IEEE CS, 2009.

[24] R. Lopez-Herrejon and D. Batory. A Standard Problem for Evaluating Product-Line Methodologies. In *Proceedings of the International Conference on Generative and Component-Based Software Engineering (GCSE)*, volume 2186 of *LNCS*, pages 10–24. Springer-Verlag, 2001.

[25] M. Mendonca, A. Wasowski, and K. Czarnecki. SAT-based Analysis of Feature Models is Easy. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 231–240. SEI, CMU, 2009.

[26] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *LNCS*, pages 419–443. Springer-Verlag, 1997.

[27] R. Tartler, J. Sincero, D. Lohmann, and W. Schröder-Preikschat. Efficient Extraction and Analysis of Preprocessor-Based Variability. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*. ACM Press, 2010.

[28] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe Composition of Product Lines. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 95–104. ACM Press, 2007.