Using Reified Contextual Information for Safe Run-time Adaptation of Software Product Lines

Sagar Sunkle
School of Computer Science
University of Magdeburg, Germany
sagar.sunkle@iti.cs.uni-magdeburg.de

Mario Pukall
School of Computer Science
University of Magdeburg, Germany
mario.pukall@iti.cs.uni-magdeburg.de

ABSTRACT

Software product lines (SPLs) is a paradigm to implement software products based on features. Contemporary SPL implementation techniques provide support for compile-time composition of features. Many approaches have been suggested for run-time adaptation of product lines which do not consider safe composition properties. This paper presents FeatureJ which enables safe compile-time and run-time composition of multiple product lines. We identify four compilation techniques necessary to achieve safe run-time adaptation of product lines in FeatureJ: (a) type checking of feature modules, (b) availability of contextual information at run-time, (c) resolving class-loading issues for multiple updated variants, and (d) an application container for executing and updating multiple variants. Accordingly, we show how an existing approach can be integrated with FeatureJ to support safe run-time adaptation.

Categories and Subject Descriptors

D.2.7 [**Distribution, Maintenance, and Enhancement**]: [Extensibility]

General Terms

Design

Keywords

Software Product Lines, Composition, Run-time Adaptation

1. INTRODUCTION

Software Product Lines (SPLs) represent a software engineering paradigm that enables creating a set of products based on the common and the variable features. SPLs enable reuse in terms of features that can be shared among software products. SPLs contain domain concepts (conceptual features) of a software system and their realization at code level (concrete features). Traditionally, the decision as to which products contain which features used to be made statically. With the emergence of dynamic software systems, this decision is no longer static and may be delayed until runtime [14]. Many researchers have assessed the value of using SPLs for such systems. Run-time composition and adaptation of SPLs has been explored in terms of dynamic product reconfiguration [14], context aware and dynamic SPLs [10].

In a parallel line of work, researchers have extended traditional SPL implementation techniques to include safe composition mechanisms for SPLs [3, 20]. Safe composition entails checking that (a) products of an SPL adhere to the specified domain constraints and (b) all selected features in the code are synthesized into a compilable product variant [3, 20]. By imposing type checking mechanisms on underlying feature representations, safe composition ensures any errors in valid products are detected at compile-time. To our knowledge, there is no approach that combines type checking of product lines with run-time composition of features to enable safe run-time adaptation.

In this paper, we take a step towards safe run-time adaptation of SPLs. We use FeatureJ which is an SPL implementation approach that differs from other approaches in its representation of features, product lines, and variants as types implemented atop the JastAdd Java compiler [7]. Both conceptual and concrete features are represented by a common semantic entity in FeatureJ [18]. Multiple product line and variant types can be defined and executed in a single FeatureJ program. We show how safe compile-time composition is supported in FeatureJ by extending semantic analyses in JastAdd. We show how FeatureJ stores the contextual information about features and how information is made available at run-time via meta-class instances of FeatureJ types which can be used to alter the structure of already defined variants. Finally, we discuss how safe runtime composition in FeatureJ can be extended to support safe run-time adaptation.

2. BACKGROUND

An SPL represents family of related programs [5]. By representing requirements of a domain as features, SPLs enable rapid customization of products. SPLs make use of feature diagrams which graphically show the relationships between the features [5]. The notepad product line (NPL) shown in Figure 1 is based on the idea that the variants of a notepad application can be obtained by selecting specific features such as creating, editing, and formatting options. An NPL variant is obtained by selecting features starting with the top level features and further down the feature di-

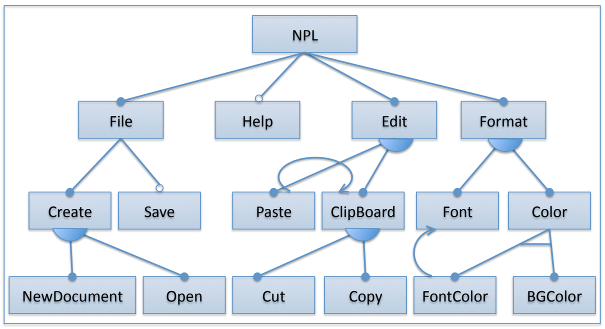

Figure 1: Feature Model of the Notepad Product Line

agram by following the feature relations for each parent feature. Based on various relations and constraints, a valid NPL product variant could be the set of features {File, Edit, Format, Color, Create, ClipBoard, Font, NewDocument, Cut, Copy}. SPLs share many characteristics with run-time adaptable systems (RTA), e.g., functionality that needs to modified and adapted can be treated as a feature [1]. In the following, we explain the relationship between SPLs and RTAs.

2.1 Run-time Adaptability and SPLs

Run-time adaptability or dynamic reconfiguration refers to software products that (a) monitor their operational context, (b) process a reconfiguration request based on resources and validation constraints, and (c) prepare and perform reconfiguration while system is running [14]. Alves et al. compare variability management in SPLs and RTAs with respect to goals, binding times, responsibility, the nature of variability models, and implementation mechanisms [1]. They observe that (a) SPLs focus on providing functional requirements whereas RTAs focus on quality of service along with functional requirements, (b) while variability is achieved at pre-compile time, compile-time, and link time in SPLs, in RTAs it is achieved at load time and run-time, (c) in SPLs, the application engineer makes variability decision whereas in RTAs this responsibility lies with the system itself, and (d) in SPLs variability models are fixed based on domain engineering while RTAs adapt variants based on reconfiguration policies. At the core for RTAs lie reflection, separation of concerns such as aspects, and component-based designs as implementation mechanisms [1].

SPLs are primarily implemented by mapping code and other related software artifacts pertaining to features and synthesizing or composing required features to obtain specific variants [2, 4, 13]. Although traditionally implemented in a static setting, SPLs are also used to facilitate dynamic reconfiguration [8, 14, 10].

2.2 FeatureJ

FeatureJ is based on the concept of feature as first-class entities [19, 18]. FeatureJ implements features as semantic entities with the properties that a) features should be represented as native first-class types or first-class entities in the host language, b) first-class features should represent the conceptual and concrete features uniformly, c) the semantics of the first-class features should be rich enough to subsume safe composition, d) the identity of features should be retained throughout the life cycle of an application to enable a manipulation at different stages of program execution, and e) first-class features should be extensible with respect to advances in feature modeling and feature composition [18].

In accordance with these properties, FeatureJ represents features, product lines, and variants as types in Java. We use JastAdd[1], a Java based compiler construction system [7] to implement FeatureJ. The type definition for productline types correlate to feature models whereas variant type definitions indicate variants obtained from these models as shown in Listing 1.

```
package testNotePadPL;
public class NotepadApps{
  productline notepadPL{
    features {
       File  :  all(Create, Save?),
       Create:  more(NewDocument, Open),
       Edit  :  more(ClipBoard, Paste),
       ClipBoard:  more(Cut, Copy),
       Format  :  more(Font, Color),
       Color  :  one(FontColor, BGColor),
       Help
     }
    all(File, Edit, Format, Help?)
    constraints {
       Paste  <-> ClipBoard,
       FontColor <-> Font
     }
  };
  variant notepadPL SimpleNotepad {
    File  = [Create],
    Create = [NewDocument],
    Edit = [ClipBoard],
    ClipBoard = [Cut and Copy],
    Format = [Font],
    Help
  };
  ...
}
```

Listing 1: FeatureJ productline and variant type declarations

FeatureJ represents both the conceptual and concrete features as types. Listing 1 shows how the productline and variant types indicating the NPL described earlier can be defined. The feature types are similarly defined in the code as block constructs that contain program elements of different granularities ranging from single statement to collection of methods [18].

3. ENABLING SAFE RUN-TIME ADAPTATION IN FEATUREJ

An important implication of representing features as types is that FeatureJ can contain multiple productline and variant types in a single program. This fact combined with the requirement that features in FeatureJ must retain their identity, necessitates four compilation techniques to ensure the

[1]http://jastadd.org/the-jastadd-extensible-java-compiler

co-existence of multiple variants and safe run-time composition and adaptation: 1) extending semantic analyses in Java compiler to address type checking of FeatureJ types to ensure safe composition of features, 2) addressing Java class-loading issues for multiple variants with separate namespaces, 3) reifying contextual information about feature, productline, and variant types, and 4) generating an application container that can instantiate multiple specific variants and execute them. In the following we elaborate each of these techniques.

3.1 Safe Composition

A FeatureJ program may contain many productline types and correspondingly many variant and feature types. The feature and variant types are declared within the context of the parent productline type. We explain the type checking of FeatureJ types in terms of two distinct processes: 1) checking the validity of feature models in productline and variant type definitions 2) type checking and error analysis of predefined variants.

Feature Model Validation

Checking the validity of feature models consists of checking productline, variant, and feature type names and validating productline and variant types.

Name resolution in FeatureJ A FeatureJ program may contain many productline types and correspondingly many variant and feature types. The name resolution of these types builds upon the Java name resolution mechanism of JastAdd. A productline type in a different package can be imported just like the regular Java types in that package. When resolving variant and feature type names, they are checked against their parent productline type. This typically consists of checking for existence of feature types used in the variant type declaration, checking for existence of feature type used in the feature containment expressions, and checking that a variant type of a given productline type refers only to the features of that productline.

Structural validation of variant types This consists of checking the structure of a variant based on its parent productline type for any inconsistencies in the feature selection based on the specified quantifiers and the inclusion and exclusion constraints. This involves validating the quantifiers on groups of features. Following this, FeatureJ checks that all inclusion and exclusion constraints are satisfied for a given variant. The structural validation and constraint resolution is carried out on per product variant basis.

Type Checking Predefined Variants

FeatureJ extendes the type checking attributes defined in JastAdd for semantically checking each of the newly generated variants for compilation errors. The compilation errors in Java involve incompatible types and expressions, incorrect pairing of modifiers, multiple declarations, reachability and normal completion errors, and exception related errors [2]. They are detected in JastAdd using the nameCheck, typeCheck, accessControl, exceptionHandling, checkUnreachableStmt, definiteAssignment, and checkModifiers attributes defined in each AST node of a Java program [7], each of which is extended in FeatureJ.

Following the type-checking of predefined variants, FeatureJ proceeds with the error analysis for multiple variants which takes into account the variant definitions in the FeatureJ program, so that FeatureJ is able to report variant specific error messages. This means that if a feature contains a semantic error but is not included in the definition of a specific variant, then error messages, if any, pertaining to this variant will not contain this error. On the contrary, if a variant is defined to include this feature then the error message is included in the error reporting for this variant. The error analysis ensures that compilation errors pertaining to specific variants due to inclusion or exclusion of features are reported accurately.

3.2 Addressing Java Class-loading issues for Multiple Variants

A Java application runs with a Java Virtual Machine (JVM) that interacts with Java classes in the .class file format. Creating first-class product lines and product variants atop a Java application indicates that for a given application many program variants may exists simultaneously. Whereas a regular Java program only runs a single version of class(es) / interface(s) comprising it, running a FeatureJ program consists of running many versions of the same class(es) / interface(s) belonging to different variants. Two scenarios may arise during the execution of a FeatureJ program which we address using the concept of namespaces and separating interface from implementation respectively as explained in the following:

Referring to different versions of the same class This scenario takes place when multiple variants exist simultaneously consisting of different versions of the same SPL class. All classes in a Java application are loaded using some subclass of the ClassLoader class. The JVM identifies a class uniquely by a combination of its qualified name and the effective class loader, i.e., the class loader that loads this class. To distinguish between the classes from different variants, they must be placed in a path that is not in the classpath of the SPL application. We achieve this by creating a separate namespace for each variant in terms of folder hierarchy with the variant name as the top folder and then all variant specific classes arranged according to packages under this folder.

Reloading modified versions of the same class This scenario takes place when multiple variants are defined in a FeatureJ program or a single variant is modified by adding or removing a feature. When a class is loaded, all classes it references are loaded recursively. A class is only loaded once and then cached in the class loader by the JVM to ensure that the byte-code of the class does not change. Reloading the class is therefore not possible using Java's class loaders. Also, the objects of classes that have exactly the same qualified name but loaded by two different ClassLoader instances are treated as objects from different classes. In order to overcome this limitation, FeatureJ generates an interface corresponding to an SPL class as the object type which is loaded once and reloads the implementing class for a specific variant. Using this arrangement, we can keep the interface

[2]http://java.sun.com/docs/books/jls/

constant while reloading the implementation class.

3.3 Reifying Contextual Information to Meta-classes

By contextual information about features, we mean the identity of features, i.e., all facts required to identify a given feature at any stage of program execution. In order to be able to use the attribute-based computational capabilities of JastAdd [7], instead of storing this information in byte-code, we store this information in the main AST during parsing of a FeatureJ program. This enables us to process the main AST of the FeatureJ program to generate ASTs of specific variants. At run-time, three classes- PL, PLVariant, and Feature, are used to represent a productline, a variant, and a feature type. These classes are conceptually similar to the meta-classes in the Groovy language[12], which is a dynamic language for the JVM.

```
import com.unimag.sag.featurej.meta.*;
public class NotepadApps {
  public static void main(String args[]) {
    PL NotepadPL= new PL("notepadPL");
    PLVariant simpleNotepad=new PLVariant(↩
        NotepadPL,"SimpleNotepad");
    Notepad n=(Notepad)simpleNotepad.↩
        getVariantObject("Notepad");
    n.setDefaultCloseOperation(JFrame.↩
        EXIT_ON_CLOSE);
    n.setSize(300,200);
    n.setVisible(true);
  }
}
```

Listing 2: Using meta-classes PL and PLVariant to access pre-defined productline and variant types

We call the PL, PLVariant, and Feature classes as meta-classes in FeatureJ in the sense that these classes are used by the FeatureJ compiler to govern the behavior of the corresponding FeatureJ types as shown in Listing 2. The information exposed using theses meta-classes includes - 1) the productline type to which current feature belongs to, 2) structural information about current feature such as its immediate scope, i.e., whether the current feature is a class body definition or method body definition, and 3) the code fragments contained in the current feature definition. This information is stored relative to each compilation unit as a feature may contain code fragments of different granularities in various compilation units.

Many queries can be executed with the objects of these meta-classes instantiated with the names of corresponding FeatureJ types as shown in Listing 2 - a) a PL object can be queried about the number features and constraints, the classes in which the constituent features are located, etc. b) a PLVariant object can be queried for its parent productline type, number and names of mandatory and optional features selected etc. c) an object of Feature class can be queried for its parent productline type, its immediate parent feature and children, the code fragments with respect to different classes in a FeatureJ program etc. The information about productline, variant, and feature types is made available to the meta-classes using the concept of an application container which we describe next.

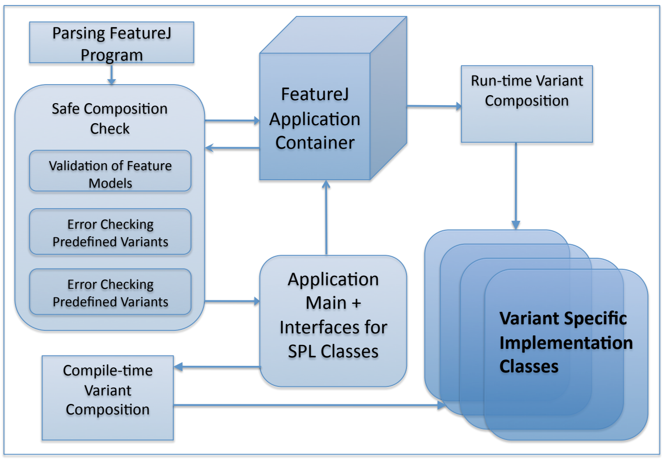
Parsing FeatureJ Program
Safe Composition Check
Validation of Feature Models
Error Checking Predefined Variants
Error Checking Predefined Variants
FeatureJ Application Container
Run-time Variant Composition
Application Main + Interfaces for SPL Classes
Compile-time Variant Composition
Variant Specific Implementation Classes

Figure 2: FeatureJ Compiler Architecture

3.4 Application Container for Multiple Variants

FeatureJ requires that one of the SPL classes be specified as an application entry point. FeatureJ uses this class and generates an application container with which to execute the FeatureJ program. The application container shown in Figure 2 can be thought of as variants deployment platform using which FeatureJ combines compilation and execution of a FeatureJ program so that multiple variant programs can be generated and executed. When input with .fjava files of a FeatureJ program, the FeatureJ compiler starts by parsing the .fjava files. Syntactic errors (if any) are found during parsing of the .fjava files. If there is no syntactic error (in FeatureJ type declarations or feature containments) then the compilation proceeds to static type checking of variants defined in the FeatureJ program. This is carried out in a manner explained in Section 3.1. If there are any errors in one (or more) variants, these errors are reported and the compilation process is halted. Otherwise, it proceeds to the generation of folder hierarchies that correspond to the variant definitions and the packages in the classes of the SPL. For example, given that NPL classes are part of the testNotePadPL package and the program contains two variant definitions (SimpleNotepad and ColoredNotepad), FeatureJ generates following directories in a temporary directory: 1) Main\testNotePadPL - For storing the interfaces corresponding to SPL classes + any pre-existing interfaces in the testNotePadPL package + the application main 2) SimpleNotepad\testNotePadPL and ColoredNotepadPL\testNotePadPL - for storing the implementations of the interfaces. To avoid class-loading problems in reloading different versions of classes from different variants, FeatureJ generates a common interface for each SPL class against which specific implementations are generated based on the variant type definitions. Using these directories to store variant specific implementation also results in separate namespace. This enables FeatureJ to avoid the Java class-loading issues as discussed in Section3.2. The implementation classes are generated by composing variant ASTs statically or dynamically as explained in the following:

Variant Composition at Compile-time If a variant type is to be composed statically, the variant AST's are composed immediately after generating the folder hierarchy and interfaces and application main as shown in Figure 2. Either of the JastAdd Backend (by transforming AST to byte-code directly) and the Sun Java compiler (by transforming the AST to Java program first and then compiling them) can be used at this stage to generate .class files for the variants. Once the .class files are generated they are stored in the corresponding PLVariant object discussed in Section 3.3, as a pair of name, and a Class class object obtained by loading these .class files. A getObject() method returns an object of the required SPL class version specific to this variant which can be used like a regular object as shown earlier in Listing 2.

Variant Composition at Run-time In FeatureJ, a statically composed variant can be altered by adding or removing features using a PLVariant object that encapsulates a pre-existing variant. A PLVariant object communicates with parent PL object which uses the main program AST to process the request regarding addition or removal of features as described in Section 3.3 and shown in Figure 2. For each addition and removal of a feature, a PL object runs safe composition check as explained in Section 3.1, because any modification can potentially introduce compilation errors. The safe composition checks are required in FeatureJ because it checks individual predefined variants instead of the entire SPL as in AHEAD [20]. If there are no errors in the resulting AST then the variant specific implementation classes are generated, compiled, and mapped. As stated earlier in Section 3.2, a separate namespace is created automatically for the modified variant so that new .class files are stored and loaded without conflict. In the next section, we show how safe run-time composition in FeatureJ can be extended to support safe run-time adaptation.

4. EXTENDING FEATUREJ WITH SAFE RUN-TIME ADAPTATION

Many approaches exist for run-time adaptation in Java which allow anticipated or unanticipated changes at varying times such as until load time or deploy-time [16, 15]. These approaches consist of using customized class-loaders [11], byte-code transformations [21], dynamic aspectual mechanisms [11, 21], dynamic update enabled JVMs [9], or a combination of these techniques. In order to extend FeatureJ's run-time composition mechanism to run-time adaptation we target an adaptation approach that is easiest to integrate with the existing FeatureJ compiler architecture.

4.1 Suitability of a Run-time Adaptation Approach for FeatureJ

Although any of the aforementioned approaches can be used for extending FeatureJ with run-time adaptation, we take into consideration the ease with which a given adaptation approach can be plugged into the FeatureJ compiler architecture. Using byte-code transformation approaches for extending FeatureJ to support run-time adaptation would require communicating the contextual information about features from meta-classes to a byte-code transformation tool. Similarly, using dynamic aspectual mechanisms such as AspectWerkz' annotations [21] would require adding another phase to the FeatureJ compiler so that contextual information about features is transformed into Java annotations over program elements. Both these mechanisms would require non-trivial extension of the FeatureJ compiler architecture.

Pukall et al. present a run-time adaptation approach that is based on implementing a) class schema changes using class renaming and b) caller update via Java HotSwap [15]. It is implemented as an Eclipse plug-in that uses JVM tool interface to obtain class information about currently running application. This approach can be easily integrated with FeatureJ's run-time composition approach for two reasons: 1) We only have to input variant specific SPL classes to the plug-in and do not have to take care of the details of byte-code transformation ourselves. 2) The class renaming technique for multiple versions of a class followed by this approach is similar to FeatureJ's approach of addressing Java class-loading issues described in Section 3.2. In the next section, we show how FeatureJ can be extended with the run-time adaptation approach by Pukall et al. [16].

4.2 Integrating Run-time Adaptation with FeatureJ

In FeatureJ, the interface generated for an SPL class represents a constant class schema. A feature exclusion that makes a method unavailable and thus results in class schema change is handled in FeatureJ as follows - a) If the FeatureJ program contains an access to a method that will be unavailable due to feature exclusion, this is treated as error and detected using FeatureJ safe composition check before generating variant specific implementation classes and the compilation process is halted. b) If the FeatureJ program does not contain an access to a method that would be unavailable due to feature exclusion, then such a variant can indeed be compiled without error because the missing method is not called. Since the interface and a specific implementation of SPL class must match so that an object of specific implementation class can be assigned to the interface as shown in Listing 2, a method that is unimplemented is inserted with a default body in the AST after FeatureJ safe composition determines that there is no error present due to an invalid access. With this arrangement, we can integrate FeatureJ with the approach by Pukall et al. [15] as described in the following:

Reconciling class renaming and separate namespaces for class schema changes In their approach, Pukall et al. rename the class to be updated and load the renamed class to indicate a different version of the same class [15]. Since, the separate namespace in FeatureJ takes care of different versions of the same class as explained in Section 3.2, we can use the FeatureJ's mechanism as it is.

Reconciling Caller Updates using Java HotSwap The next steps required in the run-time adaptation approach by Pukall et al. [16, 15] include 1) changing all calls to the instances of classes that have been updated, 2) creating an instance of the updated class, and 3) mapping the state of old callee instance to the newly generated instance. Of these, first two steps are already present in FeatureJ as this is how FeatureJ obtains an object of a variant specific SPL class as shown in Listing 2. We only need to integrate the third step in which state of the old caller object is mapped to the new instance. Once the state of old object is mapped to the new object, the new object can be used in place of old object of the original variant.

In this way, combining safe variant composition at run-time as described in Section 3.4, with the run-time adaptation approach by Pukall et al. [15], we can achieve safe run-time adaption for multiple variants and product lines in FeatureJ.

5. RELATED WORK AND CONCLUSION

Rosenmüller et al. [17] present an approach for dynamic composition of features using decorators and are currently working on run-time adaptation support, but they do not consider safe composition. Irmert et al. [11] present an approach that combines dynamic aspects in JBoss for run-time adaptation with OSGi modules that are used to create separate namespaces for services. Run-time adaptation in AspectWerkz uses class load time and run-time weaving of byte-code [21]. Dinkelaker et al. [6] demonstrate the use of meta-aspect protocol for run-time management of features by mapping first-class aspect-oriented models in running systems to dynamic feature models.

It can be observed that variability and run-time adaptability are achieved at different times during program execution [1]. In case of multiple variants and product lines this scenario is further complicated due to the fact that multiple programs may co-exist and can be composed at compile-time or run-time. This paper presented an approach for achieving this scenario at the same time assuring that variant composition and adaptation are safe.

6. ACKNOWLEDGMENTS

We thank Christian Kästner for comments on an earlier draft of this paper. Sagar Sunkle is a PhD candidate at University of Magdeburg and receives scholarship from federal state of Saxony-Anhalt. Mario Pukall's work is part of the RAMSES[3] project funded by DFG.

7. REFERENCES

[1] V. Alves, D. Schneider, M. Becker, N. Bencomo, and P. Grace. Comparitive study of variability management in software product lines and runtime adaptable systems. In *VaMoS*, pages 9–17, 2009.

[2] S. Apel and C. Kästner. An overview of feature-oriented software development. *Journal of Object Technology (JOT)*, 8(5):49–84, 2009.

[3] S. Apel, C. Kästner, A. Größlinger, and C. Lengauer. Type-safe feature-oriented product lines. Technical Report MIP-0909, Department of Informatics and Mathematics, University of Passau, 2009.

[4] D. Batory. Feature-Oriented Programming and the AHEAD Tool Suite. In *Proceedings of the 26 th International Conference on Software Engineering.* ACM, 2004.

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

[6] T. Dinkelaker, R. Mitschke, K. Fetzer, and M. Mezini. A Dynamic Software Product Line Approach using Aspect Models at Runtime. *First Workshop on Composition and Variability*, 2010.

[7] T. Ekman and G. Hedin. The JastAdd system - modular extensible compiler construction. *Science of Computer Programming*, 69(1-3):14–26, 2007.

[8] H. Gomaa and M. Hussein. Dynamic software reconfiguration in software product families. In *PFE 2003*, pages 435–435, November 2004.

[9] A. R. Gregersen, D. Simon, and B. N. Jørgensen. Towards a dynamic-update-enabled jvm. In *RAM-SE '09*, pages 1–7, New York, NY, USA, 2009. ACM.

[10] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid. Dynamic software product lines. *tse*, 41(4):93–95, 2008.

[11] F. Irmert, M. Meyerhöfer, and M. Weiten. Towards runtime adaptation in a SOA environment. In *RAM-SE'07*, pages 17–26. Universität Magdeburg, 2007.

[12] C. Kaewkasi and J. Gurd. Groovy AOP: a dynamic AOP system for a JVM-based language. In *Proceedings of the 2008 AOSD - SPLAT workshop*, page 3. ACM, 2008.

[13] C. Kästner and S. Apel. Virtual separation of concerns – a second chance for preprocessors. *Journal of Object Technology (JOT)*, 8(6):59–78, Sept. 2009.

[14] J. Lee and D. Muthig. Feature-oriented analysis and specification of dynamic product reconfiguration. In *ICSR*, pages 154–165, Berlin, Heidelberg, 2008. Springer-Verlag.

[15] M. Pukall, C. Kästner, S. Götz, W. Cazzola, and G. Saake. Flexible Runtime Program Adaptations in Java - A Comparision. Technical Report FIN-014-2009, Department of Computer Science, University of Magdeburg, Germany, Nov. 2009.

[16] M. Pukall, C. Kästner, and G. Saake. Towards unanticipated runtime adaptation of java applications. In *APSEC*, pages 85–92. IEEE, 2008.

[17] M. Rosenmüller, N. Siegmund, G. Saake, and S. Apel. Code Generation to Support Static and Dynamic Composition of Software Product Lines. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering.* ACM Press, Oct. 2008.

[18] S. Sunkle, S. Günther, and G. Saake. Representing and Composing First-class Features with FeatureJ. Technical Report FIN-017-2009, Department of Computer Science, Otto-von-Guericke University of Magdeburg, Germany, Nov. 2009.

[19] S. Sunkle, M. Rosenmüller, N. Siegmund, S. S. ur Rahman, and G. Saake. Features as First-class Entities – Toward a Better Representation of Features. In *McGPLE*, pages 27–34. Department of Informatics and Mathematics, University of Passau, Oct. 2008.

[20] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe composition of product lines. In *Proceedings of the 6th international conference on Generative programming and component engineering*, pages 95–104. ACM, 2007.

[21] A. Vasseur. Dynamic AOP and runtime weaving for Java—How does AspectWerkz address it? In *DAW: Dynamic Aspects Workshop*, pages 135–145, Mar. 2004.

[3]http://wwwiti.cs.uni-magdeburg.de/iti_db/forschung/ramses/