Passau University

Department of Informatics and Mathematics

Master's Thesis

An Extensible Compiler for Feature-Oriented Programming in Java

Author:
Sergiy Kolesnikov

First Supervisor:
Dr.-Ing. Sven Apel
Second Supervisor:
Prof. Dr. Christian Lengauer

February 21, 2011

Abstract

Feature-oriented software development (FOSD) is a paradigm for developing large software systems. Fuji, an extensible compiler, supports feature-oriented programming in Java. It does not rely on source-to-source translation but generates standard bytecode, which can be executed on any specifications-compliant Java virtual machine. This allows feature-oriented software developers to benefit from such techniques as type checking or access control, which are indispensable in standard Java compilers. Fuji is based on the JastAddJ compiler. One of JastAddJ's main development goals is easy extensibility. Thus extensibility is inherent in Fuji. This property of our compiler allows building other tools on its basis and experiment with alternative designs. In this thesis we describe the architecture and design of Fuji, as well as several extensions we implemented showing the extensibility of our compiler.

Contents

1 Introduction

Software has the tendency to grow during its lifecycle. Many application programs, for example, text processors or web browsers, start as relatively simple programs with decent functionality. In the course of further development, they grow to complex software systems. The main cause for this is the desire of the developers to satisfy the needs of as many programs' users as possible. These needs often vary radically, so that it becomes harder and harder to fulfill them all in one product. Moreover, the maintenance of the product becomes difficult because of the complexity of the design and mere amount of code to be managed.

Let us illustrate the problem on an example of a web browser. Suppose a user works with a certain browser on its desktop. It is obvious, that she would prefer to use the same browser on her smart phone because of mere convenience. The difference between computing platforms[1] may pose a great problem to the developers. In the worst case it would require vast changes in the browser code and would result in several product variants to be maintained. This example with different platforms, each having limited resources, compared to desktops, is highly relevant today, as mobile and embedded computer devices become ubiquitous.

A user may also require additional functionality, like a fully-fledged download manager, a news feed reader, a multimedia player for web page embedded multimedia. Another one may go without this additional functionality for the sake of simplicity. Thus we end up with multiple end products again. Multiply their number with the number of supported platforms and you will get the amount of product variants our imaginary web browser developers have to design, implement and maintain. During its lifecycle the browser evolves to a family of products, each offering to its users a custom-tailored solution. Such a family of products is called *Software Product Line (SPL)* [2].

We took a web browser as an example, but it is not only application software that is required to offer custom-tailored solutions. Operating system kernels or server software like database management systems often represent SPLs these days. For example, for the Linux kernel a special language and a software framework were developed to manage variability and configure single variants. Such additional development requires a great amount of extra effort, parallel to the development of the product, and the results are often project specific, i.e. cannot be easily reused by other not related projects. Thus it is a great challenge to design, develop and maintain SPLs

[1] Computing platform is a combination of hardware architecture, operating system and available software libraries / runtime environment.

with classical development methods. The need for new software engineering and programming concepts that help developing SPLs is gets stronger with every day.

There exist multiple software development concepts like stepwise and incremental software development, aspect-oriented software development, component-based software engineering, that were successfully applied to develop SPLs. All of them have their benefits and drawbacks. Feature-oriented software development is a new general paradigm that has many benefits of the mentioned concepts and aims at making the development of software product lines easier.

1.1 Feature-Oriented Software Development

Feature-Oriented Software Development (FOSD) is a development paradigm that takes into account the main problems and peculiarities of designing, implementing and maintaining SPLs. The key concept of the paradigm is the concept of a *feature*. A feature is a unit of functionality of a software system that satisfies a requirement, represents a design decision, and provides a potential configuration option [2]. On creating a product, a developer implements a set of features. Thereby she considers existing or potential customer requirements, makes some user visible designer decisions and encapsulates all these in features. When a sufficient set of features is finished, a customer can choose a subset of them and a custom-tailored end product, a *variant*, can be automatically generated for him. In terms of FOSD, a collection of all valid variants that can be generated from the set of features is called an SPL.

The development process described above has three beneficial properties, which originate for the use of the feature concept: enhanced structure, improved reuse of the components and ease of variation. Developers structure the design and code of a product with the help of features, the features can be reused in different variants, adding features to or removing them from a variant allows easy adjustment of the end product to customer needs.

Speaking of features one distinguishes between *problem space* and *solution space* [2]. In the problem space a feature describes a requirement on the resulting product and its behavior, e.g., the browser has an integrated download manager. In the solution space a feature describes how the requirements are satisfied and the requested behavior is implemented, e.g., these and these classes implement a download manger using those and those techniques. One of the main objectives of FOSD is to ensure a clean mapping of a feature in the problem spaces to a feature in the solution space through all the phases of the FOSD process, as illustrated in Figure 1. We will discuss these phases

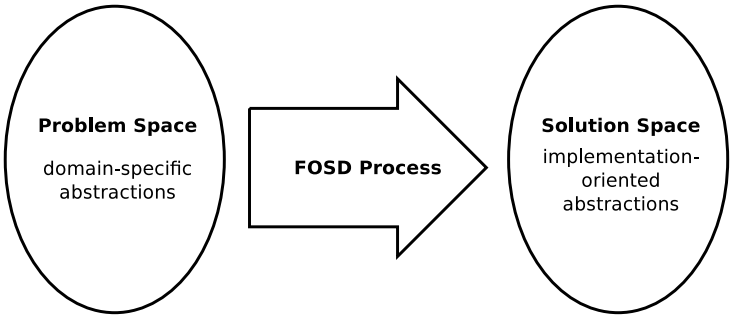

Figure 1: Problem to Solution Space mapping

in the next section.

1.1.1 Phases of the FOSD Process

The FOSD process is divided into four phases [2]:

1. Domain analysis;
2. Domain design and specification;
3. Domain implementation;
4. Product configuration and generation.

Domain analysis In domain analysis, developers deal with a problem space. They decide which features belong to the problem space, how they relate to each other, and how they can be organized. In other words they define and structure the problem space. For this task the concept of a *feature model* was introduced. A feature model describes relationships and dependencies for a set of features belonging to a certain domain. A commonly used notation for feature models is the treelike *feature diagram notation.* Figure 2 depicts a standard example, representing a product line of cars. The root of the diagram is the concept being modeled. The descendants of the root are features, which developers decided to take into the problem space. The edges between the nodes describe relations between the features. For example, feature Gasoline, describing a gasoline engine, is part of the more general feature Engine. Additional graphical elements on the edges describe further constraints, e.g., the filled circle on the edge from Car to Engine denotes, that every car variant must have an engine. The leafs of the tree represent features that will be directly mapped to the features in the solution space, because of the mentioned clean mapping property of the FOSD process. Due

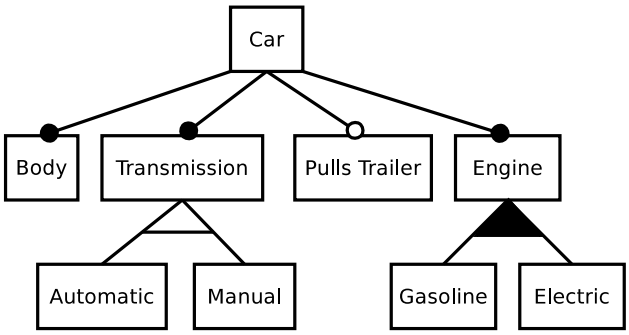

Figure 2: Feature model diagram

to the fact that feature diagrams are more comprehensible to a non-technical user, developers can present them to the customers. A customer can choose the features she wants in the end product and the developer can easily find the corresponding features in the solution space. The chosen features are then automatically composed to a custom-tailored end product.

Domain Design and Specification During domain design and specification phase, developers define the architecture of a software system. For this purpose formal specification languages like UML are often used. In the FOSD community there has not been much work in this field and proposed solutions have many open issues [2].

Domain Implementation In domain analysis, developers deal with a solution spaces. They develop features that satisfy some requirements and implement certain behavior. One of the main goals here is to achieve a one-to-one correspondence with the features defined during the domain analysis.

There exist two approaches to introduce the concept of features during the domain implement phase: the annotative approach [13] and the compositional approach [13]. The idea of the first approach is to annotate the information about the features in the code, so that different features are mixed in one code module. This can be achieved by a well-known mechanism of source code preprocessing, which is known to be one of the advantages of this approach. For example, C developers can use `#ifdef` directives in the code to annotate features and then run the C-preprocessor on it to generate variants. The disadvantage of this approach is that features are scattered across multiple code modules, and this can complicate their comprehension.

Using the second approach, developers define individual features in separate modules, which can then be composed to an end product. There are several implementations of the compositional approach. For example,

the AHEAD tools suite [6] introduces the Jak language that extends Java with feature-oriented mechanisms. Another tools suite, called FeatureHouse, implements language independent feature modularity based on attribute grammars [3]. It means that no changes must be made to the syntax of the corresponding language to make it feature aware. At the same time, the language Independence of FeatureHouse is the reason for its disadvantages. For example, a feature-oriented type system or access control mechanisms cannot be implemented, because of the lack of language specific information during the feature composition process. One of the goals of this thesis is to overcome this restriction with respect to the Java programming language. We trade language independence for more flexibility in the field of semantic error checking, in particular type checking.

Product Configuration and Generation Product Configuration and Generation is the last and the key phase of the FOSD process. The vision is that a non-technical customer can be easily involved into the configuration process, where she can express her requirements, and then a custom-tailored variant is generated automatically.

The configuration step poses a problem, because complex relations between multiple features of a large SPL can be hardly tracked by a human. Tool support in feature selection was introduced to solve this problem. There are such tools like GUIDSL[2], pure::variants[3], FeatureIDE[4] and CIDE[5].

Once we have a valid feature selection, a variant can be generated. A mechanism for variant generation that is based on superimposition was used in the course of this work and will be introduced and explained further in the text.

1.2 JastAddJ Java Compiler

This section presents the JastAdd Extensible Java Compiler (JastAddJ) [9], which we used as basis for the Fuji compiler that was developed in the course of this work. JastAddJ was built using JastAdd compiler frame work [10]. The framework utilizes the concepts like object-orientation, inter-type declarations, declarative rewrites, and attribute grammars, including support for reference attributes, nonterminal attributes and circular attributes [10]. These declarative features are the key to the extensibility of JastAddJ. The authors demonstrate JastAddJ's extensibility by first building a Java 1.4

[2] http://www.cs.utexas.edu/~schwartz/ATS/fopdocs/guidsl.html (17.02.2011)
[3] http://www.pure-systems.com/pure_variants.49.0.html (17.02.2011)
[4] http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/ (17.02)
[5] http://www.fosd.de/cide (17.02.2011)

compiler and then extending it to a Java 1.5 compiler. The specification of Java 1.5 introduced many new language constructs and concepts like generics, enums, the enhanced for-statement, autoboxing, varargs and others. Therefore, the extending of the Java 1.4 compiler was not a trivial task at all.

JastAddJ can compile programs with more than 100K LOCs. The compile times are within a factor of three compared to javac. For a research tool these absolutely acceptable characteristics.

The four main components of JastAddJ are the Java 1.4 front end and back end and the Java 1.5 front end and back end. The back ends are extensions of front ends and reuse their code. The same is true for the Java 1.5 components, which are built as extensions of the corresponding Java 1.4 components. The same approach can be used to build further extensions.

Every main component is built of four major parts: an *abstract grammar* defining the structure of abstract syntax trees (AST); *behavior specifications* defining the behavior of the objects constituting the AST; a *context-free grammar* defining how source code is parsed into ASTs; and a bootstrap program that is run by the user.

A special JastAdd language is used to specify the abstract grammar and the behavior. Based on these specifications, the JastAdd tool generates an object-oriented class hierarchy. This hierarchy is then used in the compiler to represent an AST. To parse source code an external Java-based parser must be used. JastAddJ uses Beaver[6], a Java LALR(1) parser generator, to generate the parser.

JastAddJ's components may reuse other components by just including their abstract grammars, behavior, and context-free grammars into JastAdd's compiler generation process. Figure 3 is a general illustration of this process.

Internally, JastAddJ represents a program as an AST. The AST itself is built of Java objects. The class and composition hierarchy of these objects is defined by the abstract grammar mentioned above. The behavior of the objects is defined in aspect-oriented modules (aspects) and is implemented with the help of methods. These methods represent the AST attributes. These attributes are used to solve all compilation tasks. For example, the task of finding a method declaration corresponding to a method call is implemented in the attribute `decl`, which belongs to the AST node representing this method call. A developer can write additional aspects that extend the behavior of AST nodes. Due to the usage of attributed AST and reference attributes[7] no additional data structures, like symbol tables, are needed.

[6] `http://beaver.sourceforge.net` (last access 16.02.2011)

[7] Attributes that reference other AST nodes.

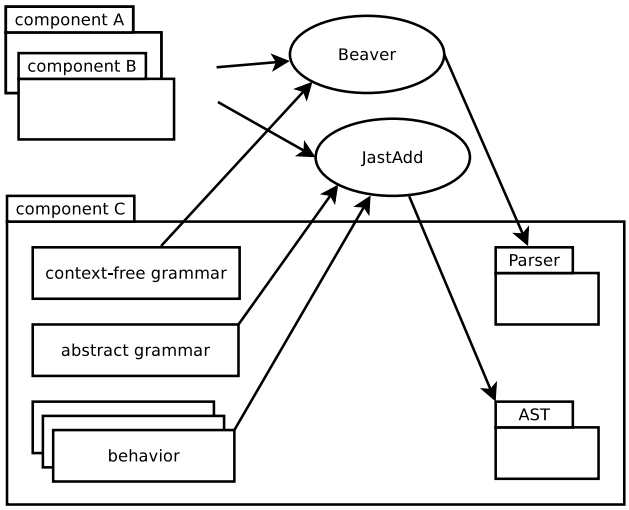

Figure 3: JastAdd generation architecture

The AST is the only data structure used in the compiler. It contains all the needed information about the compiled program and implementation of all compilation tasks.

To implement Fuji we used aspects intensively and extended JastAddJ's Java 1.4 front end. Our extension does not rely on JastAddJ's 1.5 front end, therefore the Java 1.5 extension can be easily excluded from the compiler generation process and we get a Fuji version for Java 1.4. This confirms again the high level of extensibility and modularization of JastAdd-based compilers. These properties are also inherited by Fuji.

In the remainder of this section, we illustrate the usage of aspects for extending the behavior of an AST node. As an example we take one of the tasks Fuji must fulfill during the compilation process. The task is to find out if a method declaration contains a special *original-call*[8]. For this purpose we have to extend the behavior of the AST node representing a method body, which is defined in the JastAddJ's Java 1.4 front end.

Listing 1 shows an excerpt from the front end's abstract grammar defining the composition and class hierarchy of the corresponding AST nodes[9]. The definition states that an AST node for a method declaration has children nodes representing method modifiers, return type, the name of the method, a set of parameters, and a block of code.

To implement the task of finding an original-call we define an aspect in a

[8]The meaning of the original-call is not important here and will be explained later in Section 2.3.1

[9]We left out some elements for the sake of simplicity.

```
MethodDecl ::= Modifiers TypeAccess <ID:String>
               ParameterDeclaration* Block;
```

Listing 1: Abstract grammar for method declaration

separate file. The text of the aspect is shown in Listing 2. The aspect adds the `hasOriginal` attribute to the AST node representing a method body. As mentioned above the AST attributes are represented in the AST by java methods. On traversing an AST during a compilation process, we can call the `hasOriginal()` method on a `MethodBody` node and it will return `true` if the contains an *original-call*.

```
aspect MethodComposition {
  public boolean Block.hasOriginal() {
    for (Stmt s : getStmtList()) {
      // check if the statement contains the original-call
      // if yes, return true
    }
    return false;
  }
}
```

Listing 2: Aspect implementing hasOriginal attribute.

1.3 About this Thesis

The contribution of this thesis is a successful development of Fuji compiler that supports feature-oriented programming in Java. In contrast to other feature-oriented tools, Fuji does not rely on a source-to-source transformation but is a fully-fledged compiler that produces standard Java bytecode. This opens up new possibilities in the field of semantic error checking, and type checking in particular. Moreover, its design allows easy modification for building other tools on the basis of Fuji. In the course of this work, we build several tools using Fuji's extension mechanism.

The remainder of this thesis is structured as follows:

- In Section 2, we introduce the architecture and design of our compiler and discuss its implementation details.
- In Section 3, we describe Fuji's extensions developed in the course of this work.
- In Section 4, we present the evaluation results and testing methods.
- In Section 5, we conclude the thesis.

2 Design and Implementation

This section describes the architecture, conceptual design and implementation details of Fuji. First we present the architecture of the compiler to outline its main parts and show their interconnections. In the following step we compare the structures of a typical Java project and a typical SPL. This comparison allows us refine the compiler's architecture and explain which parts of JastAddJ must be modified or extended and why. Following this top-down design strategy, we analyze each of the Fuji's essential parts and finally describe the concrete implementation.

2.1 Fuji's Architecture

Fuji is based on the JastAddJ Java compiler. JastAddJ's architecture follows the classical compiler design and consists of a front end and a back end [1]. Because of the fundamental differences in a structure of a typical Java-based SPL and a normal Java project, which are explained in Section , we extended JastAddJ's front end. We build an adapter that presents an SPL to the JastAddJ's front end as if it were a normal Java project. This allows us to reuse the most of JastAddJ's front end code to build abstract syntax trees (ASTs) for a variant. Then Fuji performs the composition of generated ASTs, which corresponds to feature composition in terms of feature-orientation. The composed ASTs are fully compatible with ASTs produced by the unmodified JastAddJ's front end and can be immediately passed to the back end for bytecode generation. Alternatively the ASTs may be given to one of Fuji's extensions for further processing. The extensions can be easily added to the compiler using provided extension mechanisms. The described architecture is illustrated in Figure 4.

2.2 Processing SPL Structure

2.2.1 Java Project Structure vs. Java-based SPL Structure

The top level element of a typical Java project is a *package*. A package can contain other packages as well as *classes* and *interfaces*. In a file system a

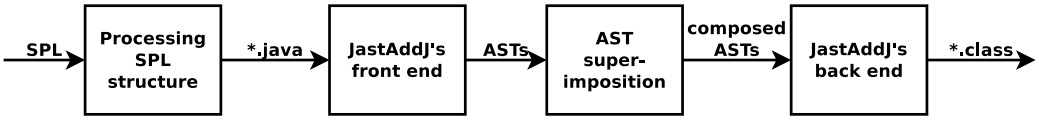

Figure 4: Fuji's Architecture

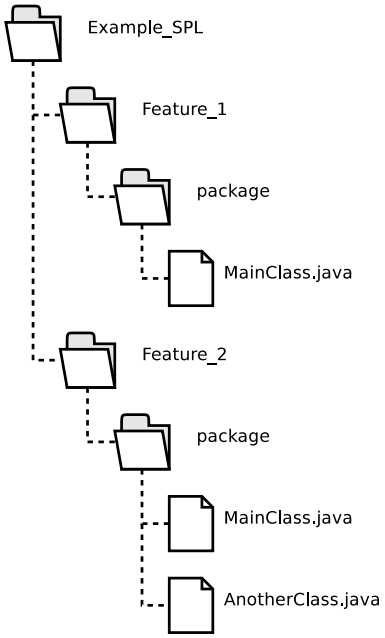

Figure 5: Structure of an example SPL

package is normally represented by a directory. Classes and interfaces are represented by files. A typical SPL implemented using the concept of *compositional approach* introduces a new structural element, namely a *feature module* [13]. This structural element encapsulates the code that implements a feature of the SPL and corresponds to a directory in the file system. Every feature module of a Java-based SPL can contain packages, classes and interfaces constituting a containment hierarchy [7]. An SLP structure can be represented by a tree, where all the feature modules of the SPL are siblings and have the same parent, namely the root of the SPL, see Figure 5. JastAddJ's front end is unaware of feature modules. Thus we implemented a modification that allowed the front end to work with Java-based SPLs.

To identify further modifications, needed because of the differences in project structures, let us consider *collaboration diagrams* [14]. A collaboration diagram can be used to visualize feature-oriented program design. Besides representing the structure of an SPL, a collaboration diagram shows relationships between the elements of an SPL. Figure 6 show a collaboration diagram of a simple SPL. From the object-oriented point of view the SPL consists of the class A and its subclass B, which are both part of the package p. From the feature-oriented point of view The SPL consists of features $F1$ and $F2$. These features decompose the core object-oriented design adding new elements, *roles*. Feature $F1$ decomposes class A into two roles, $A1$ and

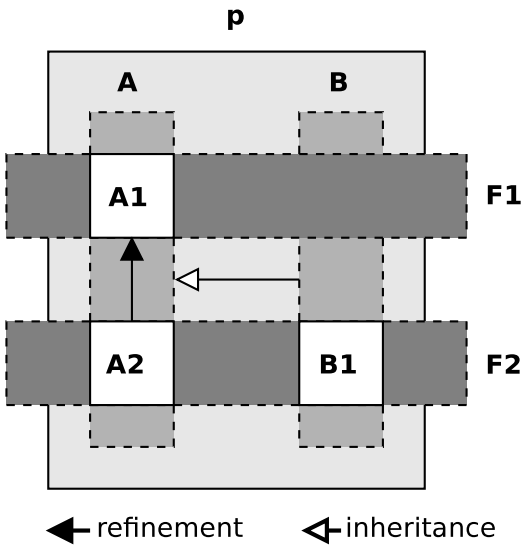

Figure 6: Collaboration diagram

$A2$. The role $A1$ is also called a *base role* and the role $A2$ is called a *refinement role*. The refinement $A2$ refines the base $A1$[10]. The class B consists of only one base role $B1$. All roles of a feature collaborate together introducing a new functionality encapsulated by the feature. In a file system one role is represented by one source file. The source files for a base role and all its refinements have the same name. Thus our example SPL will have three source files: `SPL/F1/p/A.java` (base), `SPL/F2/p/A.java` (refinement), `SPL/F2/p/B.java` (base). The presented feature-role and base-refinement relationships between structural elements of an SPL are important for the compilation process and must be implemented in Fuji.

Another important element of an SPL is a *features file*[11]. The file is used to identify a variant. It contains a list of the feature module names the variant must contain. The feature modules are listed in the order in which they should be composed.

2.2.2 Implementation of the SPL Structure Processing

In the previous section we showed, that the original JastAddJ's front-end must be extended, so that it can work with the structure of an SPL. We implemented the extension in the `fuji.SPLStructure` class. In the following the functioning of this class will be described.

[10] e.g. by adding a new method
[11] also called *expression file*

On instantiation, `SPLStructure` class gets two parameters, the pathname of the SPL root directory and the pathname of the features file. Based on this information `SPLStructure` builds an internal representation of the SPL in three consecutive steps. The resulting representation contains all the information about the SPL structure and relationships between its elements, that is required for further processing.

The steps for building the internal representation are:

1. Parse the feature file and determine the pathnames of feature modules.
2. Build role groups.
3. Calculate dependency graphs.

These are described in detail below.

Parse the feature file A feature file contains a *feature choice*, i.e. a list of the feature module names, which describes a variant of the SPL to be compiled. Besides, it specifies the order in which the features must be composed. The format of the features file is simple. The name of each feature module is specified on a separate line. The feature on the first line is the base feature. It will be composed with the feature specified on the second line and so on. Lines starting with the special character # as first non-blank character are interpreted as comments and ignored. On parsing a features file, we also check if the directories corresponding to the listed feature modules exist. It is an error, if a specified feature module does not have a corresponding directory. The result of a successful parsing process is a list of the feature module canonical pathnames. The pathnames are placed in the same order as they were listed in the features file. This list is used in the next step, as well as in the further compilation process.

Build role groups In Section 2.2.1 we showed how features decompose a class into elements called roles. In the current step all the roles belonging together[12] are detected and grouped to *role groups*. A role group is later composed according to predefined composition rules and then compiled resulting in a Java class file.

To find related roles we search through the feature module directories determined in the previous step. All the roles belonging together have identical pathnames relative to its feature module directories and can be grouped based on this property. The feature module directories are searched in the

[12] i.e. a base role and all its refinement roles

same order in which they are returned by the previous step. Thus the first role found is guaranteed to be the base role and the remaining roles are its refinements and must be composed in the order they were found.

We model a role with the `RoleGroup` class. An instance of this class contains a pathname of the base role, pathnames of all its refinement roles, selected by the given feature choice, as well as additional utility methods and fields used in the next step.

Calculate dependency graphs In the early development versions of Fuji we parsed the source code for all the role groups of the currently processed variant at once and then started the composition and compilation procedures. For the relatively big SPLs like GUIDSL[13] and Prevayler[13] this led to "out of memory exceptions" during the compilation process. The high memory consumption was caused by the way JastAddJ models an AST of the source file internally, namely, by representing each AST element as an object. For big SPLs this led to a vast number of objects to represent the AST and ended in "out of memory exceptions". To compile GUIDSL, for example, we had to set the size of the JVM's[14] memory allocation pool to 3GB. That was clearly impractical and we looked for ways to optimize memory usage.

Our solution of the memory problem is based on *dependency graphs*. Dependency graphs identify parts of the SPL's source code[15] which can be parsed and compiled separately. Formally a dependency graph $G = (V, E)$ is a directed graph, where nodes ($v \in V$) represent role groups, and directed edges ($e \in E$) represent references between the role groups. A directed edge $e = (R1, R2) \in E$ from the role group $R1$ to the role group $R2$ exists if and only if the source code of the role group $R1$ uses (references) the type defined in the source code of the role group $R2$. We also say that $R1$ *depends on* $R2$, because $R1$ can only be compiled together with $R2$. This compilation requirement follows from JastAddJ's method of operation, in particular, from the implementation of the semantic error checking.

We build a dependency graph for an arbitrary role group R as follows. First we add R to the graph. Then we add all the role groups R directly depends on. In the next step we add all direct dependencies of the groups added in the previous step. We repeat the last step until no new groups can be added to the graph. The resulting graph will contain all the role groups R depends on directly or indirectly. This is the minimal set of role groups that must be compiled simultaneously with R.

[13] see Section 4.2 for discussion of example SPLs

[14] Java Virtual Machine, an execution environment used to execute Java programs.

[15] more precisely role groups

As we are able to map from a role group to corresponding source files[16], having a role group and the corresponding dependency graph we can easily find out a subset of files, which can be parsed, composed and compiled independently from the rest.

It should also be considered that a role group may be present in several dependency graphs. This would lead to multiple unnecessary compositions and compilations of the same role group. To prevent this we add the destination directory for class files to JastAddJ's classpath by default. This ensures that a role group is composed and compiled only once. If an already compiled role group is required, the corresponding class file is loaded by JastAddJ automatically. SUN's Java 5.0 compiler does not add the destination directory for class files to classpath by default, so this Fuji's behavior must be considered non-standard. Though, we have not observed any problems that could have emerged from this modification.

Still the worst case scenario is possible, where a role group depends[17] on all the other role groups of the SPL. In this case the corresponding dependency graph will contain all the role groups of the SPL and we end up with parsing, composing and compiling all the SPL's source files at once again.

To cope with this problem we sort all the dependency graphs by the number of their nodes and start the composition and compilation process with a graph having the smallest number of nodes. The intention is to compose and compile as few role groups at once as possible. This way, processing of the graph mentioned in the worst case scenario is efficient, because most of its role groups have been certainly compiled already. Thus only few role groups will be composed and compiled, and the rest of the groups required for their compilation will be loaded from the corresponding class files.

Altogether the approach described in this section is less memory consuming than our early implementations. This is confirmed by tests. Using the described approach all the example SPL's[18] compile with 128MB[19] reserved for the JVM's memory allocation pool.

2.3 Superimposition of Abstract Syntax Trees

For composing feature artifacts[20] Fuji applies a concept of *superimposition* [5]. In this approach the composition of software artifacts is done by composing

[16] see Section 2.2.2

[17] directly or indirectly

[18] see Section 4.2 for discussion of example SPLs

[19] 128MB is the default value for SUN's Java compilers.

[20] packages, roles, types, methods, fields...

their respective substructures. For example, superimposition of two Java source files representing roles ends in one Java source file, which contains the result of composing the structural elements of the files, e.g., types, methods, fields.

The structure of a feature is represented by a *feature structure tree* (FST). All the feature's artifacts to be composed correspond to nodes in the FST. Each node of an FST has a name and a type. The notion of "type" in the previous sentence must not be confused with Java's types like `java.lang.Object`. It rather describes the syntactical category the artifact belongs to: "a method", "a field", "a compilation unit..." The nodes are divided into *terminals* and *non-terminals.*

Non-terminal nodes are inner nodes of an FST and their subtrees are subject to the recursive superimposition process. Two non-terminal nodes are superimposed by merging into one node, but only if their names and types are equal. The outcome of the superimposition is a non-terminal node with the same name and type as its source nodes. The substructure of the node is the result of superimposing the substructures of the source nodes.

Terminal nodes are leafs of an FST. Two terminal nodes with the same names and types can be superimposed only if a *superimposition rule* describing the process of their composition was defined. The outcome of the superimposition is a terminal node. Superimposition rules are application dependent and describe how the content of the terminal nodes must be composed. It must be noted that describing an artifact as a terminals is often a designer decision and is not always forced by a programming language design [13]. In our case we have to be compatible with FeatureHouse, so the terminal node definitions were taken from that implementation and will be described in the next section.

A node that does not have a partner for superimposition is just added as a child to the node resulting from superimposition of its parent.

To illustrate the process let us take a Java-based example with features $F1$ and $F2$. Suppose the features contain only one package and one file, so that we have the following artifacts in a filesystem: `F1/p/A.java` and `F2/p/A.java`. The content of the source files is listed below.

The two features are composed by superimposing their corresponding FSTs, denoted by '•'. The superimposition starts from the root and descends recursively resulting in a new FST as shown below.

The source code of the artifacts and the corresponding FSTs is shown in Figure 7.

As we can see the FST nodes representing classes are non-terminals. Thus they are merged to single non-terminal node. Each feature declares a field. The nodes representing the fields are both terminals and have dif-

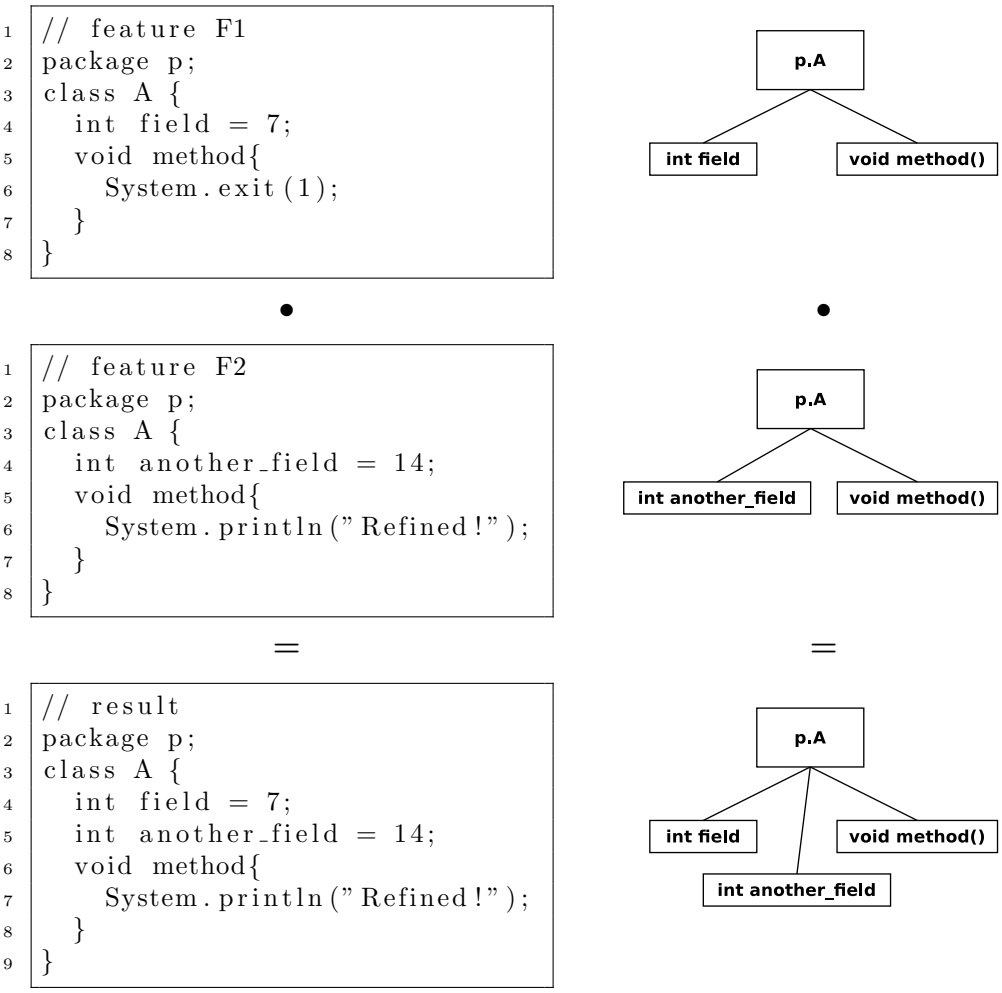

Figure 7:

ferent names. Therefore both nodes are added to the resulting FST. The feature $F2$ refines the method declared in feature $F1$ and effectively replaces its implementation. Thus the nodes representing the methods are composed according to a superimposition rule, which say that the node of the feature $F2$ must be add to the resulting FST and the node of the feature $F2$ must be left out.

Concluding the discussion of superimposition we have to note that a language of the composed artifacts must satisfy the following properties [5]:

1. The substructure of a feature must be hierarchical, i.e., a general tree.

2. Every structural element of a feature must have a name and type that

become the name and type of the node in the FST.

3. An element must not contain two or more direct child elements with the same name and type.

4. Elements that do not have a hierarchical substructure (terminals) must provide superimposition rules, or cannot be superimposed.

Java of course satisfies the listed properties [3].

Often an FST can be seen as a simplified Abstract Syntax Tree (AST) [3]. Thus we can say that two features of a Java-based SPL can be composed by superimposing the ASTs of their code artifacts. This leads to the fact that the existing Java tools (i.e. parsers) can effectively be reused in building a feature-oriented Java compiler, by making them aware of the differences described in Section 2.2.1. By superimposing two ASTs we get a new AST, which can also be processed by existing code[21], provided we have not introduced some incompatibilities. This considerations justify one more time the choice to build Fuji on the base of an existing Java compiler and explain some decisions made while designing Fuji.

2.3.1 Superimposition Rules

In this section we define a superimposition rule for each type of terminal nodes. We implemented these rules in Fuji. The rules are compatible with those implemented in FeatureHouse. FeatureHouse is a general architecture for software composition [3]. It provides facilities for feature composition of Java SPLs and uses the concept of superimposition. We also used the example SPLs from this project to test Fuji[22].

The FST node types used in these descriptions correspond exactly to the AST node types produced by Fuji's parser. Each description begins with the type of an FST node and an optional short description, then reveals how the name of the node is determined and ends with an explanation of the corresponding composition process. A terminal node may have different superimposition rules, if it is allowed in both class and interface declaration. If a rule is only applicable in either class or interface declarations this is stated in the rule. Otherwise it is applicable to both.

ImportDecl (interface). Import declarations are introduced in Java code by the `import` keyword. These terminal nodes are never composed, so their names are not important. They are just added to the resulting AST.

[21] e.g. code for bytecode generation
[22] see Section 4.2

SuperInterfaceIdList (interface). A list of superinterfaces of the given interface is introduced in Java code by the `extends` keyword. An interface can have only one such list, so the name of the node is not important. On composition, two lists are concatenated and all duplicate elements are eliminated. The resulting list is added to the target AST.

FieldDeclaration. Field declarations are treated equally for class and interface declarations. The name of a field declaration node is a concatenation of its value type and the field name. This means that two fields having the same type and name will be composed. The field declaration from the refinement AST is added to the resulting AST and the other one is ignored.

MethodDecl (interface). Method declarations in an interface are just signatures and do not provide any implementation. The name of a method declaration is a concatenation of its return type, method name and parameter types in the textual order. All the method declarations of the ASTs to be composed are added to the target AST. If a base AST and a refinement AST each contain a method with the same name, this will lead to a target AST containing two identical method declarations. According to the Java language specification an interface may not contain two identical method declarations [12]. Thus this error condition will be intercepted by JastAddJ's error checking and we don't have to implement additional checks.

ModifierList. Modifier lists may contain different modifiers like access modifiers, `static`, `final` and others. A node cannot have multiple modifier list children, thus the name is not important for this node type. Composition of two modifier lists is done by adding the refinement list to the target AST and ignoring the base list.

There is a difficulty introduced by the above rule in combination with access modifiers. If a programmer does not want to change the access modifier of a refined element[23], then she simply omits the access modifiers in the declaration of the corresponding refinement element. In Java, omitting access modifiers is syntactically equivalent to setting the accessibility of the method to *default access*[24]. This fact may cause problems during composition. Consider a situation where the base method accessibility is set to *private* and the refinement method has an empty access modifier list. The compiler cannot decide what the intention of the programmer is. Does she want to apply the accessibility defined in the base method (*private*) or does she want the accessibility of the resulting member to be extended to *default access*? By analyzing the existing SPLs we used for testing, we concluded that the common semantics behind omitting access modifiers is to preserve

[23] e.g.: a method, field or any other language construct allowing access modifiers

[24] The method is accessible only within the package it is defined in.

the accessibility defined in the base method. Thus we implemented the rule accordingly. Note that there still exists a problem in this context. Extending the accessibility of a member from *private* to *package* by refining it is not possible. Additional research is needed to find a practically sound solution for this problem, but this is outside the scope of this work.

Another consideration is that refining method access modifiers may change the type of biding the method uses. For example, a private method uses static binding. A refining, that changes method's access to public, makes it use dynamic binding. In combination with inheritance, this might lead to unexpected changes in program behavior. A programmer must be aware of this fact.

SuperClassAccess (class). This node defines the super class of the current class declaration. It is introduced by the `extends` keyword in Java source code. Its name is the name of the super class type (i.e. java.util.Vector). During composition the refinement node is added to the resulting AST and the base node is ignored.

ImplementsList (class). This list contains the names of all interfaces implemented by the class and is introduced in code by the `implements` keyword. A class may not have multiple nodes of this type. The rule for this node type is identical to that of *SuperInterfaceIdList.*

MethodDecl (class). The name of a method declaration in a class is a concatenation of its return type, method name and parameter types in the textual order. The composition process of two method declaration nodes depends on the presence of the *original-call* in the body of the refinement method. (1) If original-call is present, both nodes are added to the target AST, and the original-call is replaced by a call to the base method.[25] (2) If original-call is not present, the refinement node is added to the resulting AST, and the base node is ignored. Note that the original-call is not a special syntax. Syntactically it is a valid method call. For example, an original-call to a refined method that requires two arguments would look like `original(param1, param2);`. On composition, Fuji replaces it by the call to the base method, e.g., `base_method(param1, param2);`. Thus no modifications have been done to the standard Java syntax to incorporate this behavior. The only restriction introduced by this rule is that *original* may not be used as a method name.

ConstructorDecl (class). Constructors are a special case of method declarations in classes and are composed in the same way. The only difference is that a programmer is not allowed to make an original-call inside a constructor. The original-call is added automatically during the composition to

[25]Renaming of the base method is done here to prevent a name conflict.

every refinement constructor as the very first statement. These guaranties, that the code inside refined constructors, that normally does some important initializations, is always executed.

InstanceInitializer (class). Instance initializers are unnamed code blocks inserted directly under a class declaration. A class may have multiple such blocks. Thus the composition is done just by adding all the initializers from the base and the refinement AST to the target AST.

StaticInitializer (class). Static initializers are a special case of instance initializers. The code blocks are preceded by the `static` keyword. These nodes are composed in the same manner as instance initializers.

The above list describes all the terminal nodes for Java and their superimposition rules used in Fuji. The implementation details of the superimposition are discussed in the next section.

2.3.2 Implementation of the AST Superimposition

In Section 2.2 we described how Fuji processes the Structure of an SPL and builds a list of dependency graphs. Each such graph contains role groups that can be composed and compiled independently from role groups in other dependency graphs. As we mentioned before, a role group represents all roles constituting a class or interface, which were selected for compilation of the current variant. A role group also contains all information needed for composing its member roles. This information is primarily the paths to the source files inside the SPL structure, which implement the corresponding roles and the order in which the roles must be composed. To compose the roles we apply the concept of superimposition, which we have described above.

We implemented the composition in the `fuji.Composition` class and in some additional aspects. `Composition` class is the *context* of the *strategy software pattern* [11]. It is initialized with an instance of the `SPLStructure` class and an instance of the `Main` class. `SPLStructure` provides role groups and `Main` plays the role of the *client* in the strategy pattern. It also implements a *factory method* [11] for creating *concrete composition strategies*. Strategy pattern allows us to easily add new composition strategies that might implement different sets of superimposition rules. The implementation of concrete composition strategies will be discussed later in this section.

The method `getASTIterator()` of `Composition` returns an iterator over composed AST. Each call to the `next()` method of the iterator initiates a composition of the current role group and returns the resulting AST. The following steps describe in detail the process initiated when the client calls `next()` on the iterator:

1. Get next role group from `SPLStructure`.
2. Initialize the front end of JastAddJ.
3. Feed the source files of the role group to the front end.
4. Get ASTs for each of the source files (each representing a role) from the front end.
5. Use a concrete composition strategy provided by the client to compose the ASTs pairwise.
6. Return the resulting AST to the client.

Consider that the resulting AST is absolutely compatible with an AST expected by the JastAddJ back end. Thus the AST can be immediately checked for semantic errors and then be given to the JastAddJ back end, which will generate byte code from the AST and save it in a class file.

The last component to be discussed is the composition strategy. Each strategy implements a set of superimposition rules. The process of superimposition is implemented with the help of the *visitor pattern* [11]. A composition strategy visits the nodes of the base and refinement ASTs in parallel and applies the superimposition rules encoded in the visit-methods. Using aspect-oriented capabilities of JastAdd, we modularized the implementation of the visitor pattern in one aspect. New composition strategies can be easily added by adding the implementation of the corresponding visitor to the aspect. We used this extension capability of the pattern to implement one of Fuji's extensions (see Section 3.0.3).

2.4 Summary

Let us summarize the presented material to get a complete view of the compilation process.

The feature-oriented design of an SPL decomposes java compilation units into roles. The first task Fuji solves, while compiling a variant of the SPL, is to build a representation of the variant structure, where roles belonging to the same compilation units are gathered into role groups. To decrease memory consumption we use dependency graphs to find out sets of role groups, which can be compiled independently from the rest of the role groups. Then we start the processing of roles with a set that contains the smallest number of role groups, and reuse the already compiled groups in the following cycles. The processing itself begins with building an AST for each role. For this task Fuji uses JastAddJ's front end. In the next step the ASTs belonging

to one role group are composed to one AST. The composition is performed using the concept of superimposition. Application of design patterns allowed us to implement the composition in a clean and clear way and made the algorithm and the superimposition rules easily interchangeable. The result of the superimposition, an AST, is then passed to the JastAddJ's back end, which generates Java bytecode and saves it in a java class file. The composed ASTs may also be processed in other ways. Some examples are presented in the next section.

3 Fuji Extensions

The extensibility features of JastAddJ provided by the JastAdd framework and the design of Fuji allow easy addition of different compiler extensions. In the following we describe several extensions we implemented in the course of this master's thesis. All described extensions can be invoked using command line arguments while starting Fuji.

3.0.1 Feature-Oriented Access Modifiers

This extension implements the concept of feature-oriented access modifiers, which tackles the problem of absence of a well-defined access control model for FOP. The Java access control model cannot guarantee the encapsulation of feature code. This can lead to unexpected program behaviors and accidental type errors. To address the problem three new access modifiers were proposed: `program`, `subsequent` and `feature` [4]. These modifiers extend the Java's access control model and can be used in combination with standard access modifiers to provide missing encapsulation to feature code.

In Fuji's implementation of the concept the usage is restricted to class and interface members. The `program` modifier is the default one. If this modifier is used or no FOP modifiers are specified, then the member can be accessed by any feature. The behavior described by the extended access control model in this case is equivalent to the original one[26]. The `subsequent` modifier allows access to the member only to the code that comes from the same feature as the accessed member or any subsequent feature regarding the composition order. The `feature` modifier restricts access to the code that is situated in the same feature as the accessed member.

The novel access modifiers introduce three new keywords into the language's syntax. We modified JastAddJ's scanner and parser specifications to incorporate those modifications. Then, using aspects, we extended the type system of the compiler to implement the extended access control model.

After addition of the new access modifiers, `program`, `subsequent` and `feature` became reserved keywords. Thus these keywords cannot be used as, for example, method or field names. This restriction cannot be guaranteed for SPLs written by the authors not aware of the novel modifiers. Still we wanted Fuji to be able to compile such SPLs. The introduction of the modifiers has required changes in the scanner and parser specifications. These specifications are used by the JastAdd framework to generate our compiler. Therefore we could not implement different behavior in our compiler with the help of a command line switch, like we did it for other extensions. We

[26] i.e. the one without the additional modifiers.

provided two scanners, two parser specifications and an ANT[27] build script that allows a user to build two Fuji versions, one with and one without novel modifiers.

3.0.2 Access Analyzer

Access analyzer is an extension developed in the conjunction with the previous one. We used it to automatically analyze multiple example SPLs and identify code where the use of the feature-oriented code could be appropriate. The results were presented in [4].

The extension searches the ASTs produced by the front end for member access.[28] Then the corresponding member declarations are found. Based on the type information of the accesses and the declaration, the access analyzer calculates the most restrictive object-oriented and feature-oriented modifiers under which the identified member-accesses are still legal.

Further evaluation showed that, using the modifiers calculated by the access analyzer, a higher level of feature code encapsulation could be achieved [4].

3.0.3 Introduces and References Relations

A feature interaction is a situation where a generated variant shows unexpected behavior when two or more features are used together rather than in isolation. The standard example is a phone with two service features: call waiting and call forwarding. If a phone implements both features, it has to decide what to do with an incoming call. If it waits, the functionality of call forwarding will be broken; if it forwards, the functionality of call waiting will be broken. Feature interactions may have significant impact on the behavior of a generated SPL variant, while program designers and users may stay unaware about them and only confront the consequences. Thus feature interaction poses one of the main problems in the feature-oriented software development [8].

The extension presented here is used to deliver data for studying such feature interactions. In one mode it analyzes the ASTs of an SPL and reports which language construct[29] is introduced by which feature. In another mode it reports which language constructs are used (referenced) by which features. This data can be processed further to find out feature interactions.

Note that we analyze the ASTs for the whole SPL and not for a single variant, i.e. we inspect at all SPL features at once. To build such ASTs

[27] Apache Ant is a software tool for automating software build processes.
[28] i.e. method invocations and field accesses
[29] i.e. class, interface, method or filed

we had to introduce new superimposition rules. Usage of appropriate design patterns in Fuji's design allowed us to easily add the new set of rules and switch between them using command line arguments.

Another consequence of building an AST of the whole SPL is the increased memory consumption. We cannot apply the technique we used to decrease memory consumption[30], because it relies on reusing of the already compiled parts. However, the ASTs produced by the new set of composition rules are often erroneous regarding Java type rules and, therefore, uncompilable. For example, multiple declarations of the same method may be present in one class, if multiple features have alternative implementation of the same method. The increased memory consumption might be problematic for analyzing big SPLs, because it may reach 3GB and more.

3.0.4 Source-to-Source Translation

This extension translates the code of a given variant into normal Java code that can be compiled by a normal java compiler. Fuji works here in the way similar to that of FeatureHouse, which does source-to-source translation too, but with one major difference: Fuji will report all the semantic errors it has found in the code and their exact positions in the *original* source code. Whereas using FeatureHouse the user will get compiler error messages pointing to the translated code and not to the original one.

We implemented this extension because JastAddJ's back end produced inconsistent bytecode for several of our example SPLs. The inconsistencies appeared due to several bugs in the back end[31] and resulted in unexpected run time exceptions during program execution.

[30] See Section 2.2.2, Calculate dependency graphs.

[31] See our bug reports #39 and #41 at `http://bugs.jastadd.org` (last access on 7 Feb. 2011)

4 Evaluation

In this section we evaluate our compiler. To test Fuji we wrote 33 mini-SPLs that were used for output-based testing and regression tests. Besides, we chose 10 SPLs of different size from the SPL collection of the FeatureHouse project to test compatibility with FeatureHouse and evaluate Fuji on bigger SPLs with real-life functionality, e.g., a UML editor, a data compression library. In the following we describe the obtained results.

4.1 Output-Based Testing

Every time we implemented new functionality in Fuji, e.g., a new superimposition rule or the feature-oriented access modifiers, we wrote a mini-SPL that implemented a use case invoking this new functionality. The mini SPLs fulfilled a dual purpose. On the one hand they performed as unit tests that tested the introduced functionality. On the other hand they acted as regression tests that have the purpose to discover regressions. *Software regression* is a defect in an existing functionality introduced by a freshly added functionality, patch or any other code change.

The testing method we used is output-based. We compared error messages of the compiler or messages produced by the compiled variants of mini-SPLs with expected output. If the produced output matched the expected one, the test was considered as passed.

For regression tests to be effective, they have to be run systematically after every more or less significant code change. Otherwise, it would be difficult to determine which code modification caused a particular regression. With growing number of tests, running every one of them manually after every code change became more and more time consuming. To automate the test execution and result evaluation, we wrote a testing frame work. The frame work is implemented in Bash scripting language and uses standard GNU tools like `find` and `diff`. Every mini-SPL has a *run-scripts* and a file containing expected output. The run-script knows how to build and run a variant and which compiler command line arguments to use. The run-script also saves the output messages in a file. A master script executes all run-scripts and compares produced messages with the expected output. The result of executing the master script is a report that presents all tests and their results.

For every bug that we found in the compiler and that was not covered by existing tests, we created a new mini-SPL before fixing the bug. In combination with regular regression tests, it allowed us to reduce the amount of defects in the code in a systematic way. Most of the mini-SPLs cover multiple use cases and tests, so that the amount of actual tests cases is greater than

the number of mini-SPLs.

Combination of techniques like test-driven development, regression tests and disciplined refactoring allowed us to introduce new functionality that required design changes and keep the negative impact on existing functionality as low as possible. Using output-based testing we could successfully find and fix bugs in our code.

4.2 Example Software Product Lines

We took the example SPLs form the SPL collection of the FeatureHouse project. The SPLs differ significantly in the number of features and lines of code (LOC). They were developed by various authors and present a broad range of real-life functionality. Altogether they build a representative pool of test cases. In the following we give a short description of each SPL and acquired test results.

GUIDSL GUIDSL is a product line configuration tool. It can read feature models in GUIDSL format and allows configuring and analyzing different variants based on the feature model. This application was developed using FOP techniques, in contrast to some of the example SPLs that were created through refactoring of existing Java applications.

Violet Violet is a simple UML editor. It was created by decomposing an existing Java application of the same name into features. Violet was the first application where we observed inconsistent bytecode produced by JastAddJ's front end. The defect resulted in a `java.lang.VerifyError` during execution of the variant with all features enabled. Updating JastAddJ to the last SVN snapshot solved the problem; apparently the bug was solved in the meantime.

Prevayler Prevayler is an open source object persistence library for Java. It was created through refactoring an existing application. Fuji's semantic error checks found multiple bugs in this application, which can be classified into 2 types. The first type encompasses bugs caused by wrong access modifiers. For example, one of the classes declared a constructor with default access, but code from another package tried to instantiate it. Public access to the constructor was necessary in this case. Fuji found 4 defects of this type. They were all fixed by extending the accessibility of the members appropriately. The second type includes compile errors caused by unreachable code, e.g. a statement after `throw` clause. Fuji found 2 defects of this type.

They were fixed by commenting out the unreachable code. Another problem with Prevayler SPL was that JastAddJ's back end produced invalid byte code. Updating to the last SVN snapshot did not solved the problem. So, we investigated the problem further and found a bug[32] in the JastAddJ's front end. Definite assignment check for local variables, as specified in Chapter 16 of The Java Language Specification [12], produces false negatives under certain circumstances. Therefore usage of an uninitialized variable in Prevayler code was not detected properly, that resulted in inconsistent bytecode. We found the corresponding declaration of the variable and initialized it with `null`. This fixed the bug.

PKJab PKJab is an instant messaging client for XMPP (Extensible Messaging and Presence Protocol) protocol, also known as Jabber protocol. Variants of this SPL could be successfully compiled using Fuji.

ZipMe A pure Java zip compression library for Java mobile applications. It was created by refactoring an existing library of the same name. We wrote a test application that zips and unzips a dummy file, to be able to test the variants compiled with Fuji.

TankWar A rael-time strategic war game. The SPL uses Java multimedia capabilities. Variants for mobile platforms are provided by this SPL. Variants of this SPL could be successfully compiled using Fuji. The application was developed from scratch using FOSD.

AJStats AJStats is a tool for collecting statistics of AspectJ programs. It is used to explore how aspects are being used in current AspectJ projects, and was developed from scratch using FOSD. Variants of this SPL could be successfully compiled using Fuji.

Notepad A text editor SPL. The basic variant represents a minimal scratchpad without save functionality. Save, undo, clipboard, print and so on are implemented in separate features. The most feature-rich variant is a fully-fledged editor for styled text. This product line was written from scratch using FOSD.

[32] http://bugs.jastadd.org/cgi-bin/bugzilla/show_bug.cgi?id=41 (last access on 7 Feb. 2011)

SPL	Domain	Features	LOC	Compilation (sec.)
GUIDSL	SPL configuration tool	26	10084	56.54
Violet	UML editor	88	7194	20.18
Prevayler	object persistence lib.	6	5268	18.62
PKJab	Jabber client	8	3373	12.53
ZipMe	ZIP library	13	3520	07.09
TankWar	strategic game	15	2830	07.34
AJStats	software analysis tool	20	13226	05.78
Notepad	text editor	10	891	05.59
GPL	graph library	9	646	04.53
EPL	expression evaluation	11	105	01.36

Table 1: Example SPL Statistics

GPL Graph product line is a family of classical graph applications. Each variant can model a certain graph type, e.g., directed weighted graph, with unique set of features, e.g., search algorithm and cycle checking.

EPL A small SPL written from scratch that does expression evaluations. Variants of this SPL could be successfully compiled using Fuji.

Table 1 summarizes the information about the example SPLs and specifies for each SPL the name, application domain, number of features in the variant used for testing, LOC in the variant and time required for compilation.

Using the example SPLs to test Fuji showed its strength, namely, the ability to find semantic errors in the SPL code, but also revealed several defects in JastAddJ compiler Fuji is based on. JastAddJ is not actively supported anymore. Thus it is difficult to say if the defects will be eliminated in near future. Nevertheless, there are no comparable alternatives to JastAddJ at this time.

4.2.1 Testing Environment

The compile time measurements were carried out under a GNU/Linux OS, using OpenJDK Runtime Environment (IcedTea6 1.9.5). We used Bash build-in command `time` to measure the runtime of the processes. JVM bootstrapping took thereby about 0.1 sec. The testing machine was a Fujitsu Celsius W480 workstation with Intel® Xeon® 2.93GHz (4 Core), 8GB DRR3 SDRAM (1333 MHz) and Serial ATA-300 HDD.

5 Conclusion

In this thesis we presented Fuji, an extensible compiler for feature-oriented programming in Java. We described its architecture, design principles and implementation details, which were strongly influenced by such feature-oriented concepts like compositional approach in implementing software product lines and superimposition of feature structure trees.

We successfully showed the extensibility of our compiler by describing several extensions we implemented in the course of this work. The extensions utilize type system, access control model, abstract representation of the compiled program, and were hardly implementable in alternative tools that use source-to-source translation.

During development of the compiler we applied elements of test-driven development and wrote multiple mini software product-lines, as well as an automated testing framework. These were also used to run regressions tests, after we introduced new functionality. The described approach helped us minimize the amount of defects in the code, and keep all the components working until the end of the development. The compiler was successfully tested on 10 software product-lines differing in the number of features, LOC and application domain.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 2006.

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

[3] Sven Apel, Christian Kästner, and Christian Lengauer. Featurehouse: Language-independent, automatic software composition. In *In Proc. Int'l Conf. on Software Engineering*, page 2009. Universitätsbibliothek Passau; Fakultät für Informatik und Mathematik. Mitarbeiter Lehrstuhl/Einrichtung der Fakultät für Informatik und Mathematik, 2009.

[4] Sven Apel, Sergiy Kolesnikov, Jörg Liebig, Christian Kästner, Martin Kuhlemann, and Thomas Leich. Access control in feature-oriented programming. *Science of Computer Programming*, In Press, Corrected Proof, 2010.

[5] Sven Apel and Christian Lengauer. Superimposition: A language-independent approach to software composition. In Cesare Pautasso and Éric Tanter, editors, *Software Composition*, volume 4954 of *Lecture Notes in Computer Science*, pages 20–35. Springer, 2008.

[6] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling stepwise refinement. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 30(6):2004, 2004.

[7] Don S. Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, jun 2004.

[8] Muffy Calder, Mario Kolberg, Evan H. Magill, and Stephan Reiff-Marganiec. Feature interaction: a critical review and considered forecast. *Comput. Netw.*, 41(1):115–141, January 2003.

[9] Torbjörn Ekman and Görel Hedin. The jastadd extensible java compiler. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 1–18, New York, NY, USA, 2007. ACM Press.

[10] Torbjörn Ekman and Görel Hedin. The jastadd system –- modular extensible compiler construction. *SCP*, 69(1-3):14–26, 2007.

[11] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, Boston, MA, January 1995.

[12] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[13] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in software product lines. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 311–320, New York, NY, USA, 2008. ACM.

[14] Yannis Smaragdakis and Don Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Trans. Softw. Eng. Methodol.*, 11(2):215–255, April 2002.

Eidesstattliche Erklärung: Hiermit versichere ich an Eides statt, dass ich diese Bachelorarbeit selbständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und dass alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind, sowie dass ich die Bachelorarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Sergiy Kolesnikov,
Passau, 21. Februar 2011