Feature Oriented Programming in Groovy

Guilherme Assis
Federal University of Minas Gerais (UFMG), Brazil
ghdeassis@gmail.com

Gustavo Vale
University of Passau
Germany
vale@fim.uni-passau.de

Eduardo Figueiredo
Federal University of Minas Gerais (UFMG), Brazil
figueiredo@dcc.ufmg.br

ABSTRACT

Software Product Line (SPL) aims to reuse code and other artifacts in order to reduce costs and gain agility. Feature Oriented Programming (FOP) is a technique to develop SPLs that aims to improve the modularity and flexibility of feature code. The basic idea of FOP is to decompose software into smaller pieces, called features, so they can be composed according to the needs of each customer. Currently, Groovy programming language has no tool or framework that supports the implementation of FOP-SPLs. Groovy is a programming language that has been growing in popularity in recent years. Given this scenario, this work proposes and evaluates G4FOP, a light way to develop SPLs using Groovy. G4FOP extends FeatureHouse which is a framework for software composition and FOP. A preliminary evaluation shows that G4FOP covers Groovy grammar. We also demonstrate by an example that G4FOP is suitable to develop SPLs. G4FOP is currently integrated to the official FeatureHouse repository.

CCS Concepts •**Software and its engineering → Software notations and tools** → Development frameworks and environments

Keywords Groovy, Feature Oriented Programming, FeatureHouse, Software Product Line

ACM Reference format:

Guilherme Assis, Gustavo Vale, and Eduardo Figueiredo. 2017. Feature Oriented Programming in Groovy. In *8th ACM SIGPLAN International Workshop on Feature-Oriented Software Development, Vancouver, Canada, October 2017 (FOSD'17)*, 10 pages.

DOI: 10.1145/3141848.3141851

FOSD'17, October 23, 2017, Vancouver, BC, Canada

ACM ISBN 978-1-4503-5518-6/17/10..$15.00
https://doi.org/10.1145/3141848.3141851

1 INTRODUCTION

Software companies normally maintain similar products to address particularities of different costumers. In this context, Software Product Line (SPL) is an interesting strategy because it claims to increase productivity, software quality, and to reduce time to market [24]. The idea behind SPLs includes managing a platform with different features satisfying specific needs of a particular market segment [3]. A feature represents an increment in functionality or a system property relevant to at least one stakeholder [16]. Therefore, products derived from an SPL share common features and differ themselves by the set of selected features. Such variations are called SPL variability.

To develop SPLs, we can use different techniques, such as preprocessors [14], aspect-oriented programming [17], and feature-oriented programming [9]. In particular, Feature-Oriented Programming (FOP) is a programming technique for modularity and systematic reuse of features. In FOP, the source code of each feature is grouped into packages (features) and, when a feature is selected to compose a product, its code is merged in; for instance, by the superimposition mechanism [9]. The superimposition mechanism consists of composing the source code of the selected features in a pre-established order. There is a considerable support for FOP in different languages, such as Java, C, C#, Haskell and XML [6, 11]. However, to the best of our knowledge, there is no work that provides support to other popular programming languages, such as Groovy and Python [25].

We believe that providing support for these languages is desirable not only for practitioners, but also for researchers. For instance, since Groovy is an optionally typed language, practitioners can explore this Groovy characteristic to develop SPLs. The SPL research community, on the other hand, can analyze and compare different development approaches for software modularity and reuse. For instance, researchers can perform quantitative and qualitative studies aiming to compare typed and optional typed languages in the FOP-SPL context.

Therefore, the main goal of this work is to provide a light way to develop FOP product lines using Groovy. To do that, we propose G4FOP (Groovy for Feature Oriented Programming). We decided to support Groovy programming language due to three main reasons. First, it has a substantial growing community in the last 3 years [28]. Second, it is similar to Java and, hence, it is easier for Java developers to learn Groovy and Java libraries can be easily integrated to Groovy code [2]. Third, it is an optional

typed language, providing more flexibility to programmers [19]. This last point is particularly interesting for further work aiming to investigate the benefits and drawbacks of either using or not using types in SPLs.

To achieve our goal, we extended FeatureHouse [4], a well-known framework for FOP. The proposed extension consists on creating four main FeatureHouse artifacts for Groovy: parser, syntax mapper, composition rule checker, and feature structure tree (FST) printer. The parser gets the input code and it builds the syntax tree. The syntax mapper maps the syntax tree into the FST. The composition rule checker checks if the grammar is correct and, finally, the FST printer records the composed FST on disk. Our extension is evaluated in two ways: a demonstration of use with an SPL example and a systematic verification of the grammar constructions based on a guide for Groovy development.

The main contribution of this work is G4FOP. We expected that G4FOP fills the gap as an optional typed language to develop feature oriented software product lines. Additionally, since G4FOP extends a well-known framework, we expect that it is going to be soon available in an integrated development environment (IDE) for SPLs, such as FeatureIDE [15]. In fact, we already managed, with the FeatureHouse maintainers, to integrate G4FOP code into the official FeatureHouse repository.

The remainder of this paper is organized as follows. Section 2 sets the context of this study by providing background to related topics, such as software product lines and Groovy. Section 3 presents G4FOP and details how we generate the parsers and how we compose features. Section 4 describes and presents our example of use and the evaluation of Groovy grammatical constructions. Section 5 discusses limitations and threats to validity of this research work. Finally, Section 6 concludes this paper and presents suggestions for future work.

2 BACKGROUND

This section presents some concepts to understand the rest of this paper. Section 2.1 gives an overview of feature oriented software product lines. Section 2.2 briefly discusses software composition in the light of FeatureHouse concepts. Section 2.3 introduces Groovy and explains the reasons for selecting this programming language.

2.1 Feature-Oriented Software Product Lines

Software Product Lines (SPL) can be defined as a strategy to develop software systems that have similar features managed in a unique platform [24, 30]. A feature represents an increment in functionality or a system property relevant to some stakeholders [16]. Therefore, features are continuously being tested and reused, increasing the software quality and reliability. In addition, new features can either implement new functionalities in the SPL or be used to enhance new and existing products.

The benefits of SPLs can be seen, normally, after the third product because initial investments are required from a company to develop and maintain the SPL [10, 26, 20]. These investments include the time of learning and training developers and maintainers of a company, for instance. On the other hand, expected benefits include improvements in productivity, software quality, and modularity of source code [20].

Feature Oriented Programming (FOP) is a technique to develop SPLs that favors the systematic application of the concept of features in all phases of the software life cycle. In FOP context, the life cycle consists of analyzing, designing, implementing, customizing, debugging, and evolving SPL artifacts [3]. Fig. 1 demonstrates the phases of a feature oriented programming process. These four phases are divided in domain and application engineering and problem and solution space. In the first phase, the stakeholders should analyze the domain to identify its features and relationships. In the second phase, features from the target domain should be implemented. The third phase is the application engineering. In this phase, we should define the requirements to compose the SPL products. Once the features are defined, the products are built by combining the selected features in the fourth phase [15]. In the next section, we see how the selected features can be combined into a product by FeatureHouse.

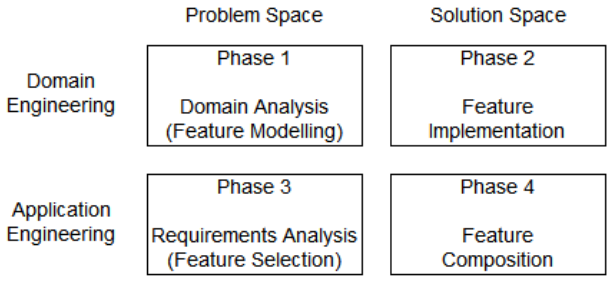

Figure 1: The FOP process.

2.2 FeatureHouse

FeatureHouse is a framework for FOP based on a language independent model and a plugin for integration of new language artifacts [5]. It supports managing software variability and composing products of an SPL. Software composition is the process of building systems from a set of features [7]. This purpose is to favor the reuse, customization, and maintenance of large software systems. A popular approach to software composition is called superimposition.

Superimposition is the process of composing software artifacts of different components by merging their corresponding substructures. The two component trees are composed recursively through the composition of nodes of the same level, counted from the root, with the same name and type. With superimposition, two trees are composed by merging their corresponding nodes, starting from the root and progressing recursively up to the leaves. Two nodes are composed to form a new node when their parents, if they exist, have been composed. In other words, two nodes are composed when they are on the same level and have the same name and type. The new node receives the same name and type as the two composed ones. If two nodes are composed, the process continues with their children. If a node is alone (i.e., there is no other node for it to be composed), it is added as the child of the new tree that was created [7].

FeatureHouse can be used to compose software in several programming languages, such as Java, C, C++, C#, Haskell, Alloy, JavaCC, XHTML and XMI/UML. It is basically composed of two tools: FSTComposer and FSTGenerator [5]. *FSTComposer* is based on a general software artifact structure model, called Feature Structure Tree (FST). Packages, classes, methods and other structures can be represented in a FST as nodes. The internal nodes are called non-terminal nodes, while the leaf nodes are called terminal nodes [4].

Fig. 2 represents a FST example with a package, a class, a method, and two fields. *FSTGenerator* receives a grammar written in FeatureBNF format, which is similar to Backus-Naur form (BNF). BNF was introduced by John Backus and Peter Naur in the 1960s as a formal notation to describe the syntax of a language format [21]. It generates some artifacts that are used by the FSTComposer tool to compose the software from the SPL configuration [5].

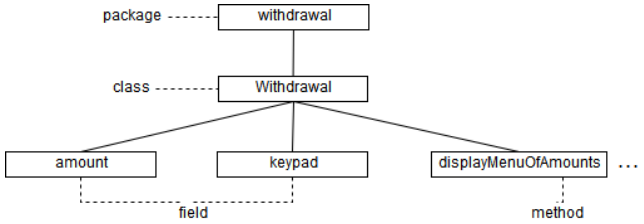

Figure 2: Example of Feature Structure Tree.

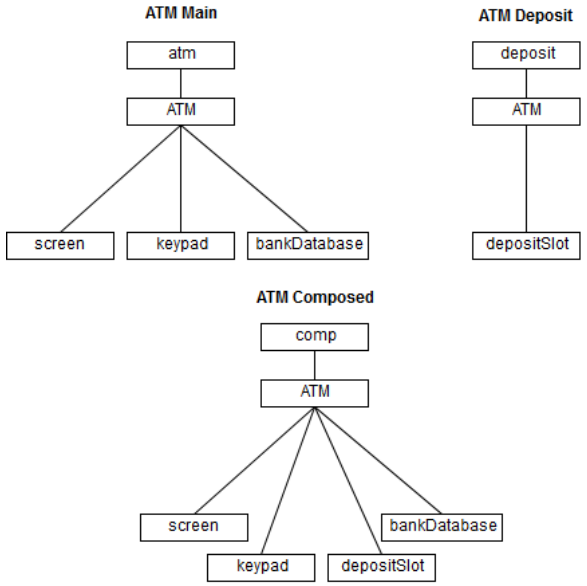

Figure 3: Superimposition of FSTs.

Fig. 3 shows the superimposition process of two FSTs from an SPL. This SPL represents a typical bank ATM with features like deposit and withdraw. The first FST represents *Main* feature and the second FST represents *Deposit* feature. The FST (bellow these two FSTs) represents the composed FST from these two features. Fig. 4 complements Fig. 3 by showing the related code resulted of this example. In particular, it illustrates how the "atm" package is composed with the "deposit" package in a third package "comp" created by the composition. Nodes called "ATM" are composed with the nodes of the same name in the other tree, and their sub-trees are composed recursively.

```
1 package atm
2 class ATM {
3     Screen screen
4     Keypad keypad
5     BanckDatabase banckDatabase
6     ...
7 }
```

●

```
1 package deposit
2 class ATM {
3     DepositSlot depositSlot
4     ...
5 }
```

=

```
1 package comp
2 class ATM {
3     Screen screen
4     Keypad keypad
5     BanckDatabase banckDatabase
6     DepositSlot depositSlot
7     ...
8 }
```

Figure 4: Code example of superimposition.

2.3 Groovy

Groovy is an optionally and dynamically typed language for the Java platform [2]. This language was not designed to replace the Java, but rather to complement it [18]. Additionally, due to its similarity with Java, Groovy is seamlessly integrated with Java libraries and, hence, Java programmers can easily learn and use it [19]. Groovy has several features that were inspired by other programming languages, such as Python [24], Ruby [27], and Smalltalk [28]. Moreover, it can also be used as a script language [19]. The use of types is optional in Groovy. Therefore, it combines the usefulness of dynamically typed languages, such as Ruby and Python, with the stability and power of Java [18].

According to the TIOBE ranking [29], in September 2015 Groovy was the 37th most used programming language in the world. A year later, it was already the 16th of the list. Therefore, it jumped 21 positions in one year. Fig. 5 shows the growth of Groovy in the last 10 completed years. This recent growth may

have been due to two main reasons. First, Groovy was incorporated by the Apache Software Foundation in 2015. Second, it may have benefited by the popularization of Web frameworks, such as Grails [1].

Although much of Groovy grammar is similar to Java, one language is not a subset of the other. For example, some of the similarities of the two languages are the mechanisms of packaging, throwing exceptions, instantiating objects, and declaring and using generics and annotations. We can cite four points as being the main values aggregated by Groovy in general: (i) easy to access to Java objects through new expressions and operators, (ii) the fact that the language allows different ways of creating objects using literals, (iii) new control structures, and (iv) the introduction of new data types along with their operators and expressions [18].

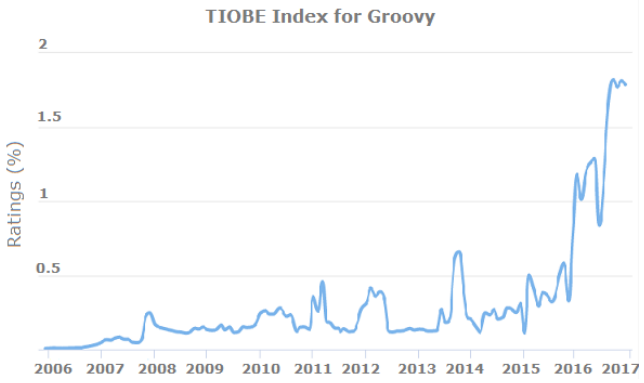

Figure 5: Groovy usage growth in recent years [29].

3 G4FOP

This section proposes an extension of FeatureHouse, named G4FOP (Groovy for Feature Oriented Programming), to support Groovy programming language for SPL development. Section 3.1 provides an overview of how we developed G4FOP. Section 3.2 discusses details about the used grammar. Sections 3.3 and 3.4 present the two tools, called FSTGenerator and FSTComposer, for generating Groovy parser and composing feature, respectively.

3.1 Overview of Groovy Extension

The idea behind FeatureHouse is that, although software artifacts vary widely from language to language, the process of composing software artifacts through superimposition is very similar. For any new language that will be integrated into the framework, we need to provide four main artifacts. The first artifact is a *parser* and its classes representing the language syntax tree. The second one is an *adapter* that maps the syntax tree to the Feature Structure Tree (FST). The third artifact is a *grammar* with a set of language specific composition rules and the fourth one is a *printer* to write the superimposed FSTs on disk [4]. From a grammar, the FSTGenerator tool generates the other artifacts [5]. With all artifacts in hand, FSTComposer can read a file with the SPL features that have been selected to a valid product configuration.

Fig. 6 shows the phases of the G4FOP creation process. To create G4FOP, we first create the language grammar in FeatureBNF format in order to compose features implemented in Groovy using FeatureHouse. This grammar has its structures annotated with @FSTTerminal and @FSTNonTerminal for terminal and non-terminal nodes, respectively.

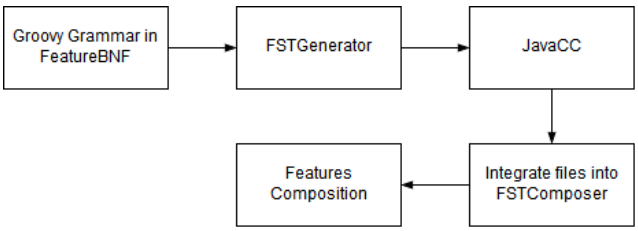

Figure 6: Phases of G4FOP creation process.

The second step was to execute the FSTGenerator tool, which is part of the FeatureHouse framework, passing as parameter the grammar written in FeatureBNF format. FSTGenerator creates a printer, which is responsible for writing the resulting files of the composition into the disk. FSTGenerator also converts the grammar from FeatureBNF format to the JavaCC format [23]. JavaCC is a parser and lexical parser generator. It reads the description of a language and generates the code, written in Java, to read and analyze that language. The grammar converted to JavaCC is represented by a file in the ".jj" extension and it is the input the third step. When executing JavaCC in the converted grammar, other six Java files are generated: "TokenMsgError.java", "GroovyParserConstants.java", "CharStream.java", "Token.java", "GroovyParser.java", "GroovyParserTokenManager.java", and "ParseException.java". These files should be integrated into the FSTComposer to be used in the moment of the composition.

The fourth step is to use FSTComposer to compose software products written in Groovy. To do that, we provide to the composer a file with the path of all selected features representing a valid configuration. Such selection can be done manually by checking the constraints between them, or semi-automatically using FeatureIDE [15]. FeatureIDE is an Eclipse-based development environment that supports all phases of FOP in the development of SPL [15] and it checks all cross-tree constraints. At the end of the fourth step, a valid configuration should be provided in a file with ".features" extension. Hence, we provide all required data to FeatureHouse compose features fulfilling our fifth step.

3.2 Building the Grammar

The creation of a grammar to be integrated into the FeatureHouse was done using the FeatureBNF format [16] (Section 2.2). FeatureBNF has added some artifacts to the BNF so that the programmer can annotate the grammar of the language with attributes. Fig. 7 shows an example of annotation in a Java-like grammar. The two main ways to annotate the grammar with FeatureHouse are to place an annotation @FSTTerminal or

@FSTNonTerminal. The first annotation defines that node is terminal and the second one defines that the node is not terminal. By default, all production rules that do not receive annotations are defined as terminals. Therefore, the advantage of using @FSTTerminal is to be able to specify the name that appears on the FST node and the composition order of two nodes [5].

```
1  @FSTTerminal
2  ImportDeclaration :
3     "import" [ "static" <NONE> ]
4     Name [ ImportPackage ] @! [ <SEMICOLON> ]
5  ;
```

Figure 7: Excerpt from Groovy grammar in FeatureBNF.

Most of Groovy grammar processing to integrate the language into FeatureHouse is performed by using JavaCC. A grammar written in FeatureBNF consists of two parts. The first part contains instructions for JavaCC, especially for lexical analysis, since the tokens of the language are declared in this part. The second part is separated from the first one by the word "GRAMMARSTART", and it consists of the syntactic constructions of the language [16].

Fig. 8 shows the production of the closure structure of Groovy grammar as an example. In this figure, there is the creation of the production called `ClosureExpression`. Two more productions are called inside it: `ClosureParameterInitializer` and `Expression`. `ClosureParameterInitializer`, in line 6, represents the initialization of a closure, which is a comma-separated list of parameters. "<IDENTIFIER>" represents any identifier used to name an attribute. "<COMMA>" represents a comma and "<ARROW>" an arrow (->). These symbols are defined in the first part of the grammar. Expressions inside brackets are interpreted as optional. If expressions are enclosed in parentheses with an asterisk in the end, that means it can repeat itself several times.

```
1  ClosureExpression :
2     "{" [  <IDENTIFIER> ]
3     (ClosureParameterInitializer) *
4           [ <ARROW> ] (Expresion) * "}";
5
6  ClosureParameterInitializer :
7     <COMMA> <IDENTIFIER>;
```

Figure 8: Production of closure structure.

The grammar of the Java language in FeatureBNF was used as inspiration for writing the Groovy grammar. The Java grammar was already integrated in FeatureHouse and it was used because the languages are similar. However, several adaptations were necessary, since Groovy programming language brings some differences for Java. It was necessary to add some keywords to the grammar. For example, we removed the semicolon at the end of *as* and *in* commands, since it is optional in Groovy. It was also necessary to change the variable typing to optional, since in Groovy it is possible to declare a variable using the word *def* without defining its type at compile time.

New ways of declaring some of the language structures, such as *array*, *map*, and *for*, have also been created. For example, in Groovy an array from 0 to 20 can be defined with the following command: (0..20).toArray(). Hence, we created new structures, such as the closure call. Closure is a block of code that can receive parameters, return a value, and be associated with a variable [2]. It was also necessary to make the grammar recognizes the Groovy structure called *expand*, which is a meta-class capable of expanding its state and behavior [18].

Finally, we enabled the grammar to support new operators that are present in Groovy, such as *spread* and *safe*. Fig. 9 exemplifies the spread operator which performs an action on all components of a collection. In this example, all words in the list would be converted to uppercase. Fig. 10, on the other hand, shows an example of using the *safe* operator to avoid throwing `NullPointerException` because it checks if the object is null before attempting to retrieve the requested attribute.

```
1  def words =
2  [ 'Feature', 'Software', 'Grammar' ]
3  words*.toUpperCase()
```

Figure 9: Groovy language spread operator.

```
1  def user = User.get(1)
2  def username = user?.username
```

Figure 10: Grooy language safe operator.

3.3 Generating Parsers

The idea behind the FSTGenerator is to abstract the most complex part of writing a grammar in JavaCC by writing of a grammar in the FeatureBNF format. With the Groovy grammar in FeatureBNF, FSTGenerator generates the grammar in the JavaCC format and a printer. The FSTGenerator is called using the following command: `java -jar fstgen.jar groovy fst.gcide groovy.jj`, where "fstgen.jar" is the FSTGenerator in Java executable format, "fst.gcide" is the grammar written in FeatureBNF and "groovy.jj" is the file to be created by FSTGenerator.

Fig. 11 shows an excerpt of Groovy closure structure code that was generated by FSTGenerator from the production written in FeatureBNF. This structure was presented in Fig. 8. By this example, it is possible to see that it is much easier to write productions in FeatureBNF and executes FSTGenerator (Fig. 8) than to write directly the code of Fig. 11.

With the "groovy.jj" file, we run JavaCC by passing this file as a parameter, through the following command: `javacc.bat groovy.jj`. The lexical analyzer and the parser generates the six files described as follows. "TokenMsgError" is a simple error class, which is used for errors detected by the lexical parser. "ParseException" is another class of error used for errors detected by the parser. "Token" is a class representing tokens, in which

each object has its identification code and the string represented by this token as attributes. "GroovyParserConstants" is an interface that defines the tokens used by the language, providing these data to the lexical analyzer and to the parser. "CharStream" is a class that delivers characters to the lexical parser which, in turn, is represented by the GroovyParserTokenManager class. The parser is represented by the class "GroovyParser" [22].

To make FSTComposer uses the generated files, we created a folder called "groovy" within the FSTGenerator project. We have to place the following files inside this folder: "groovy.jj", "GroovyParserTokenManager", "SimplePrintVisitor" "Groovy-Parser", and "GroovyParserConstantes". The other files do not need to be copied, since they are generic. In this way, they are already configured within FeatureHouse. Once these steps are completed, the FSTGenerator is configured for Groovy and we just needed to configure the FSTComposer to access these artifacts when the files are composed (next subsection).

```
1  FSTInfo ClosureExpression (
2        boolean inTerminal ) : {
3     Token first=null, t; FSTInfo n;
4  }
5  { { first=getToken (1): productionStart (
6      inTerminal ) ; } (
7      "{" [  <IDENTIFIER> ]
8      (n=ClosureParameterInitializer (true) {
9          replaceName(n) ; } ) *
10         [ <ARROW>] (n=Expression (true) {
11         replaceName (n); } ) * "}" {
12            return productionEndTerminal (
13                  "ClosureExpression", "-",
14                  "-", "Replacement",
15                  "Default", first, token );
16         }
17 ) }
18 FSTInfo ClosureParameterInitializer (
19         boolean inTerminal) : {
20    Token first=null , t ; FSTInfo n;
21 }
22 { { first=getToken (1); productionStart
23           (inTerminal); } (
24      <COMMA> <IDENTIFIER> {
25         return productionEndTerminal(
26              "ClosureParameterInitializer",
27              "-", "-","Replacement",
28              "Default", first , token);
29      }
30 )
33 }
```

Figure 11: Closure JavaCC production.

3.4 Composing Features

FSTComposer is the tool of the FeatureHouse framework responsible for composing the selected features of an SPL. This tool has a functionality called "original". This functionality is responsible for merging the body of two methods with the same name, but belonging to different features. Therefore, "original" is a reserved word. When composing two features, if both have a method with the same name, FSTComposer searches the body looking for the original word. If it finds, this word is replaced by the body of the other method. If it does not find it, the composition is given by replacing one method by the other [5].

This way of merging two method bodies is one of the composition rules of FeatureHouse, and is called method *override*. There are six more composition rules used: constructor concatenation, field specialization, union of implements lists, modifier specialization, substitution, and content concatenation. *Constructor concatenation* adds the declarations of one constructor to the declarations of the other. For example, if two constructors are being composed, their declarations are concatenated into one. *Field specialization* assigns an initial value to a field if it has not had one before. *Union of implements lists* links the list of interfaces that the classes implement, excluding duplicates. *Modifier specialization* specializes modifiers. For example, if two attributes of the same name are being composed, one of type "Integer" and the other of type "int", the result of the composition is "int", since it is the more specific type. *Substitution* replaces one terminal node with another. Finally, *content concatenation* concatenates content represented by text from two terminal nodes [5]. It is the FSTComposer itself that, at the time of the composition, analyzes case by case and checks the best rule to be applied.

4 EVALUATION

We designed the G4FOP evaluation in two steps. First, we analyzed the coverage of Groovy grammar constructs based on a guide [8]. Second, we developed an SPL in Groovy to verify the practical use of G4FOP. These two ways are described in Sections 4.1 and 4.2. Section 4.3 summarizes the lessons learned.

4.1 Evaluating Grammatical Constructions

To ensure that we widely covered the Groovy constructions and structures, we first looked at the Groovy documentation [2]. After that, we used a Groovy guide of the site "Learn X in Y Minutes" [8] to evaluate the coverage of Groovy grammar constructions. This site brings together a set of quick tutorials and examples of the programming language structures, not only for Groovy, but also for several programming languages.

As an example, Fig. 12 shows a part of that guide regarding the closure structure of Groovy. Line 1 declares the closure "clos" that prints on the screen the phrase "Hello World!". Line 4 executes such closure. Line 7 defines the closure "sum". This closure receives two parameters and prints the result of the sum between them. Line 8 runs the closure created on the previous line.

Fig. 13 shows another example, showing how to add a meta-class behavior of Groovy. In this example, Line 1 adds a new method to the String class, called "testAdd". When this method is called from a String instance, line 2 is executed, printing "test content" in the console. Lines 5 and 6 show how to use the new behavior of String. Line 5 creates a new instance of String, called "x", with the value "test". Line 6 uses the safe operator to access this method, printing "test content" in the console.

```
1  def clos = { println "Hello World!" }
2
3  println "Executing the Closure :"
4  clos()
5
6  //Passing parameters to a closure
7  def sum = { a, b -> println a + b }
8  sum (2, 4)
```

Figure 12: Code example closure structure.

```
1  String.metaClass.testAdd = {
2            println "test content"
3  }
4
5  String x = "test"
6  x?.testAdd()
```

Figure 13: Code example adding meta class.

In fact, this step of our evaluation consists of implementing each construction of "Learn X in Y Minutes". Hence, we checked if G4FOP covers all constructions used in the guide, since it includes most Groovy constructions. Based on the results of this evaluation step, we identified and fixed any existing problem in G4FOP grammar. Therefore, for all constructors evaluated, we verified that our extension is in conformance with the Groovy grammar.

4.2 Demonstration of Use

The goal of developing G4FOP is to allow the development of SPL in Groovy. To demonstrate that we achieved our goal, we modeled and implemented an SPL in Groovy using G4FOP. That is, we demonstrate a practical way to develop an SPL in FOP using Groovy. The project we use consists of a common automatic teller machine (ATM).

Fig. 14 shows the feature model of our project. This ATM SPL simulates a keypad and a screen, where users can enter the information and see their actions. As Fig. 14 shows, the possible transactions are: withdraw, deposit and check the balance inquiry. The ATM system has a database to manage such transactions and the user can have two types of account (checking and saving). In addition, a product of the ATM SPL can have a cash dispenser or a deposit slot. However, when the withdraw transaction is included in a product, the cash dispenser feature must also be selected. That is, the first feature requires the last one, as shown in Fig. 14. Similarly, when the deposit transaction is selected, so must be the deposit slot feature.

We implemented the ATM SPL in about 1.0 KLOC of Groovy code. Even with this small SPL example, 66 products can be derived. For instance, we can create a product with all features (full configuration) or a product that does not allow the user to deposit money. Each function was extracted for a separate feature organized according to its hierarchical structure, and respecting the constraints.

The code of each feature was packed into a folder called as the feature name. For instance, all parts of code regarding deposit is placed on a folder named "Deposit". We also created files with ".features" extension that represent valid product configurations. For instance, *full-Product.features* represents a product with all features. Hence, after we have developed the features and selected a valid configuration, we run FeatureHouse with G4FOP. FeatureHouse is responsible for composing the code of a product in a new directory with the name of the product. In our example, the directory is named "full-Product".

Fig. 15 shows an example of a part of Groovy ATM implementation. First, the abstract class "Transaction" of feature *Transaction* is presented. This class has the abstract method called "execute". This method must be implemented by the features that depend of *Transaction*, such as *Withdraw*, *Deposit*, and *BalanceInquiry* (see Fig. 14). After the implementation of the Transaction class, we present an excerpt of the implementation of class Deposit from feature *Deposit*. This class extends Transaction and implements the "execute" method, being responsible for receiving the deposit of the ATM user. This example highlights how the code of two dependent features should be to be composed by G4FOP. In this example, Transaction is a non-terminal node and Deposit represents a terminal node. This code is composed like the superimposition presented in Fig. 4.

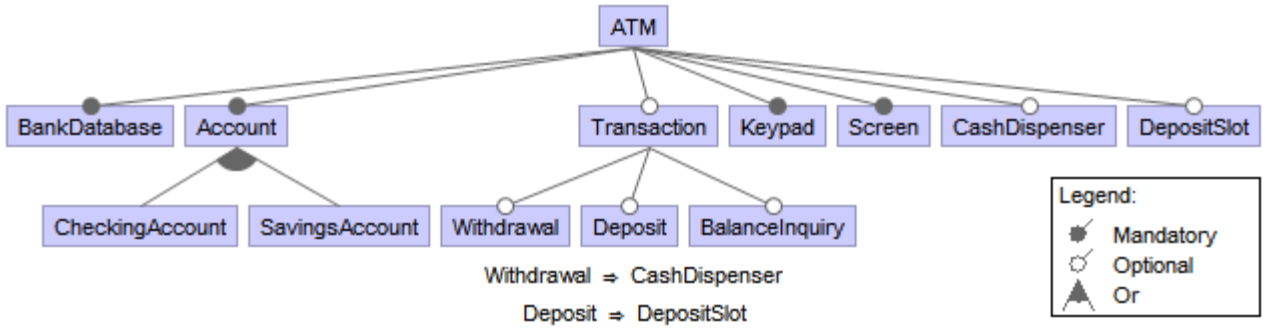

Figure 14: ATM SPL Feature Model.

Feature Transaction

```
1  public abstract class Transaction {
2    private int accountNumber
3    private Screen screen
4    private BankDatabase
5    abstract public void excute();
6  }
```

Feature Deposit

```
1  public class Deposit extends Transaction {
2    private double amount
3    private Keypad keypad
4    private DepositSlot depositSlot
5    @Override
6    public void execute () {
7      BankDatabase bankDatabase =
8            getBankDataBase()
9      Screen screen = getScreen()
10     amount = promptForDepoistAmount();
11     if (amount !=0) {
12       screen.displayMessage(
13          "\n Please insert a deposit" +
14          "envelope containing" )
15       screen.displayDollarAmount(amount)
16       screen.displayMessageLine(".")
17       boolean envelopeReceived  =
18          depositSlot.isEnvelopeReceived()
19       if (envelopReceived) {
20         screen.displayMessageLine(
21           "\n Your envelop " +
22           "has been received")
23         bankDatabase.credit(
24           getAccountNumber(), amount)
25       } else {
26         screen.displauMessageLine(
27           "\n You did not " +
28           "insert an envelope, so" +
29           "the ATM has canceled" +
30           "your transaction." )
31       }
32       } else {
33         screen.displayMessageLine("\n" +
34           "Cancelling transaction… " )
35       }
36     }
37   }
38 }
```

Figure 15: Excerpt of Deposit and Transaction features in Groovy.

To verify if G4FOP together with FeatureHouse was composing products correctly, we generated and tested 4 different products. In all cases, the composition was exactly as it should be. Therefore, we verified that G4FOP respects Groovy grammar and feature composition mechanisms in all evaluated scenarios.

4.3 Lessons Learned

We deal with some challenges in this work, such as writing the grammar, learning FeatureHouse usage and developing G4FOP extending FeatureHouse. We highlight some of these challenges below.

Writing a grammar in Groovy was challenging because, among other reasons, it is an optionally typed language. Hence, in several points of the grammar it was necessary to treat the existence or not of the type of a variable, for example. In addition, the language has several ways to abbreviate some commands, bringing an extra complexity to the grammar writing.

Since G4FOP is an extension of FeatureHouse, the complexity of using it is not so big, especially if the developer is already familiar with the framework. If the few constraints reported in this work are followed and the settings are made properly, the composition of features of an SPL developed in Groovy should occur without problems.

In addition to G4FOP, this work also helps the creation of possible future new extensions of FeatureHouse, since it details the steps taken to integrate Groovy into the framework. Thus, an extension to a new language could be created from the steps described in this paper.

5 THREATS TO VALIDITY

Even with the careful planning, some factors may have affected our research results and findings. We discuss limitations, threats and respective treatments to reduce the impact of these factors as follows. Groovy is a programming language that can be used as a scripting language. To achieve that, the source code does not have to be inside classes and methods. However, the superimposition mechanism to develop feature-oriented SPLs refine methods. Hence, all source code, except variable declarations, must be inside methods. As a result, the source code should be inside classes for G4FOP to work properly. These two grammar limitations hinder the use of G4FOP as a scripting language.

There are two main threats to the validity of this work. The first is related to our demonstration of use. Although ATM SPL may not reflect a real SPL due to its reduced size, we build a feature model with several variability possibilities, such as with optional and cross-tree constraint relationships. The second threat to validity is related to the scope of our grammatical evaluation. It is possible that we did not cover all Groovy grammar constructors. To reduce this threat, we checked more than one documentation source as we described in Section 4.1.

6 RELATED WORK

We did not find any paper that implements FOP support for Groovy. Hence, we considered some frameworks that, in general, support FOP as related work. These frameworks include previous versions of FeatureHouse, AHEAD, FeatureC++, and rbFeatures as we discuss below.

The main related work are those corresponding to the creation of the framework FeatureHouse, since G4FOP is an extension of it. FeatureHouse emerged from the proposal of generic approach of software composition to FOP, which resulted in the creation of FSTComposer tool [7]. Later this approach was generalized, creating FeatureHouse and the tool FSTGenerator, which one

facilitated the integration of new programming languages to the framework [5].

Before the creation of FeatureHouse, an extension of the FSTComposer tool for C# programming language was created, allowing compositions to be made using superimposition [26]. Similarly, an extension was also created for XML artifacts [11].

It is also important to mention the Color IDE (CIDE) approach, which is part of FeatureHouse. CIDE investigates how resources can be extracted from legacy software, and today it is implemented in the tool called gCIDE, which is part of FeatureHouse [16].

AHEAD uses a similar approach of FeatureHouse, but it is limited to Jakarta (jak) programming language [9]. Jakarta consists of a superset of Java. In other words, with AHEAD developers can develop SPLs writing a Java code (until version 4). FeatureC++ [6] is a C++ programming language framework to support FOP. In FeatureC++, features are implemented by mixin layers. A mixin layer consists of a collaborative set of mixins, where each one implements a class fragment. Mixin layers can be presented in several classes.

Finally, rbFeatures [13] is a domain-specific language framework for FOP. A domain specific language provides the appropriate abstractions and notations for a domain. In rbFeatures, there are four first-class entities (Feature, Feature-Model, ProductLine, and ProductVariant) that are high abstraction level and represent a complete SPL and its variations. Features in rbFeatures are implemented through mixins, similar to FeatureC++ [13].

Given all these frameworks that allow us to develop SPLs using different programming languages we opted to extend FeatureHouse. Our extension, G4FOP, is different from other extensions and frameworks because we provide support to develop feature-oriented SPLs using Groovy.

7 CONCLUSION AND FUTURE WORK

This work proposed a way to compose software from an SPL developed with Groovy programming language. We first present Groovy showing its growth and importance in the current scenario of the software market. The growth of Groovy can be justified by the popularization of Web frameworks, such as Grails. In addition, Groovy is Java-like, which makes the learning curve for Java programmers smooth. The fact that Groovy was designed to run on the Java Virtual Machine is a great advantage, since you can take advantage of existing libraries and Java code. In addition to providing various facilities relative to syntax compared to Java, Groovy also provides interesting mechanisms such as functional programming support.

After the presentation of Groovy, we discussed the concepts like FOP and SPL, showing how to model and implement an SPL in a way that facilitates the composition of its features. In this context, we presented the FeatureHouse framework, and how its tool set can be used to compose software using the composition paradigm called superimposition. We also presented the possibility of integrating new languages into the framework by writing a grammar in the FeatureBNF format. It was presented the level of granularity used in writing Groovy grammar, which considers as terminals only methods and attributes.

We explained the extension of the FeatureHouse framework for Groovy language, how the grammar was written, which was based on Java due to the syntax similarity of the languages. However, several differences and additions were required. Steps were taken to generate the artifacts needed to integrate the language into the FeatureHouse, and later we showed how to compose software using the extension. A demonstration of use was made in an SPL that was implemented in Groovy, showing how it was modeled and implemented following the FOP concepts. An evaluation of Groovy language structures supported by extension was also demonstrated. Therefore, we could see that G4FOP works properly and respect Groovy grammar as well as feature oriented mechanisms, such as feature composition and cross-tree constraints.

The main contribution of this work was to provide the community a way to implement SPL using Groovy language. For this, an extension of a recognized academic framework (FeatureHouse) was created, which uses a well-known paradigm of software composition (FOP). We proposed G4FOP without creating many restrictions in Groovy code to be composed. Such contribution permits researchers explore dynamic typed languages in the context of SPLs. G4FOP can be a starting point to quantitative and qualitative comparisons and analyzes. Such studies can compare performance, quality and stability of typed, non-typed and dynamic typed languages, for instance. Furthermore, we submitted a pull request to FeatureHouse maintainers include G4FOP and, as result of this request, all G4FOP code is now integrated to the official FeatureHouse repository [12].

As future work, we aim to remove the limitations imposed by the Groovy extension that was created. For instance, restrictions related to the composition of scripts (Section 5). Removing these limitations would make the grammar and FeatureHouse support scripting. That is, it would not require all code to be within a class and within a method, except for variables for the latter point. We also plan to perform a comparative study on the use of (optional) types in software product lines. For this study, we aim to use G4FOP for the SPL implementations.

There are several other languages that are being widely used, such as Ruby. A possible future work would be to integrate more languages into the FeatureHouse, in order to popularize it even more. In order to ensure that the created extension works properly, another future work would be to evaluate the use of G4FOP in a real and large-scale SPL implemented in Groovy. And, we plan to compare the software quality aspects of SPLs developed using Groovy (G4FOP) and other programming languages.

ACKNOWLEDGMENTS

This work was partially supported by CAPES, CNPq (grants 290136/2015-6 and 424340/2016-0), and FAPEMIG (grant PPM-00651-17).

REFERENCES

[1] Apache. 2017. The grails framework. http://www.grails.org/, 2016. Accessed in July/2017.

[2] Apache. 2017. The groovy programming language. http://www.groovy-lang.org/. Accessed in July/2017.

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

[4] S. Apel, C. Kästner, and C. Lengauer. 2013. Language-independent, automated software composition: the FeatureHouse Experience. IEEE Transactions on Software Engineering (TSE) 39(1): 63-79.

[5] S. Apel, C. Kästner, and C. Lengauer. 2009. FeatureHouse: Language-independent, automated software composition. In Proceedings of the 31st International Conference on Software Engineering (ICSE), pages 221-231.

[6] S. Apel, T. Leich, M. Rosenmüller, and G. Saake. 2005. FeatureC++: feature-oriented and aspect-oriented programming in C++. In Proceedings of 4th International Conference on Generative Programming and Component Engineering (GPCE). Springer.

[7] S. Apel and C. Lengauer. 2008. Superimposition: A language independent approach to software composition. In International Conference on Software Composition, pages 20-35. Springer.

[8] A. Bard and R. Alcolea. 2017. Learn groovy in y minutes. https://learnxinyminutes.com/docs/groovy/. Accessed in July/2017.

[9] D. Batory, J. Sarvela, and A. Rauschmayer. 2004. Scaling step-wise refinement. IEEE Transactions on Software Engineering, 30(6):355-371.

[10] D. Dikel, D. Kane, S. Ornburn, W. Loftus, and J. Wilson. 1997. Applying software product-line architecture. Computer, 30(8):49-55.

[11] J. Dörre and C. Lengauer. 2009. Feature-oriented composition of xml artifacts. master's thesis, Department of Informatics and Math., University of Passau.

[12] J. Liebig. FeatureHouse repository. 2017. https://github.comm/joliebig/featurehouse. Accessed in July/2017.

[13] S. Günther and S. Sunkle. 2009. Feature-oriented programming with ruby. In Proceedings of the First International Workshop on Feature-Oriented Software Development (FOSD), pages 11-18.

[14] Y. Hu, E. Merlo, M. Dagenais, and B. Laguë. 2000. C/C++ conditional compilation analysis using symbolic execution. In Proceedings of the International Conference on Software Maintenance (ICSM), pages 196-206.

[15] C. Kästner, T. Thum, G. Saake, J. Feigenspan, T. Leich, F. Wielgorz, and S. Apel. 2009. Featureide: A tool framework for feature-oriented software development. In Proceedings of the 31st International Conference on Software Engineering (ICSE), pages 611-614.

[16] C. Kästner, S. Trujillo, and S. Apel. 2008. Visualizing software product line variabilities in source code. In Software Product Line Conference (SPLC), pages 303-312.

[17] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. 1997. Aspect-oriented programming. In European conference on object-oriented programming (ECOOP), pages 220-242.

[18] D. Koenig, A. Glover, P. King, G. Laforge, and J. Skeet. 2007. Groovy in action, volume 1. Manning.

[19] V. Layka, C. M. Judd, J. F. Nusairat, and J. Shingler. 2013. Beginning Groovy, Grails and Griffon. Springer.

[20] F. vd Linden, K. Schmid, and E. Rommes. 2007. Software product lines in action: The best industrial practice in product line engineering.

[21] M. Marcotty and H. Ledgard. 2012. The world of programming languages. Springer Science & Business Media.

[22] T. S. Norvell. 2007. The javacc tutorial.

[23] P. K. Oracle. 2016. Java compiler compiler (javacc) - the java parser generator. https://javacc.java.net/.

[24] K. Pohl, G. Böckle, and F. J. van Der Linden. 2005. Software product line engineering: foundations, principles and techniques. Springer Science & Business Media.

[25] Python. 2017. https://www.python.org/, 2017. Accessed in July/2017.

[26] A. von Rhein. 2008. Feature-orientierte program-mierung mit superimposition in C#.

[27] Ruby. 2017. https://www.ruby-lang.org, 2017. Accessed in July/2017.

[28] Smalltalk. 2017. http://www.smalltalk.org, 2017. Accessed in July/2017.

[29] Tiobe index. 2017. http://www.tiobe.com/tiobe-index/ Accessed in August/2017.

[30] G. Vale, D. Albuquerque, E. Figueiredo, A. Garcia. 2015 "Defining Metric Thresholds for Software Product Lines: A Comparative Study". In: 19th International Conference on Software Product Line (SPLC), pages 176-185.