Rhizome: A Feature Modeling and Generation Platform

Guozheng Ge, E. James Whitehead, Jr.
Dept. of Computer Science
University of California, Santa Cruz, USA
{guozheng, ejw}@cs.ucsc.edu

Abstract

Rhizome is an end-to-end feature modeling and code generation platform that includes a feature modeling language (FeatureML), a template language (MarkerML) and a template-based code generator. A software designer creates feature models using FeatureML by selecting and defining design choices. These design choices can be automatically associated with code templates and interpreted as parameter values for code generation. The code generator then replaces markers embedded in the code templates with dynamically generated code blocks to produce source code.

1. Introduction

Rhizome [1] is a feature modeling and code generation platform for automatic feature-to-code generation for software product lines [2, 3] that share a common platform and vary by individual feature configurations. This platform includes a feature modeling language (FeatureML), a template language (MarkerML) and a template-based code generator. Using FeatureML, a software designer creates a feature model that contains feature design choices selected from a product line design space. Then, using the Rhizome code generator, these design choices are automatically implemented at the source code level. The templates used for code generation are developed in advance by platform developers: they study existing software product variants and capture code structure variability using tags written in MarkerML. At code generation time, feature design choices in a feature model are interpreted as parameter values. The code generator then uses these parameter values to create code blocks and resolve variability in template file content by replacing markers with these generated code blocks.

The Rhizome platform is a simple, flexible and practical solution that connects high level feature models with code level implementations. Instead of taking the MDE model transformation approach such as QVT [4] to capture domain knowledge using models and meta-models, we rely on human developers to understand domain knowledge and express variability directly using tagged source code. High level feature design choices are parameterized and interpreted at appropriate locations using recurring code generation patterns. Examples of code generation patterns include generating code for conditional branches, collection operations, global string replacement, etc. These patterns are the core of the template-based code generation employed by the Rhizome platform. Rhizome is also designed in a modular way so that new discovered patterns can be easily integrated into the code generator. Hence, the Rhizome platform can be used to automatically implement any textual language on any platform as long as templates have been developed for that language.

Several existing research efforts and commercial products provide functionality similar to Rhizome, as described below. Pure::variants [5] is a commercial tool for variant management in software product lines. It first asks the user to define a family model and relate this model to assets such as components, files, tests, documentation, etc. Then, the user defines a feature model that contains a specific collection of design choices. With this information, pure::variants automatically selects and configures related assets based on the feature model. Compared to the pure::variants configuration management approach, Rhizome takes a code generation approach that directly embeds feature-to-code associations in template files that are ultimately transformed into source code. Rhizome also has finer granularity control for variability expressions—it is up to the variability tags to decide what needs to be generated: a file, a method, a few lines, or simply a string.

XVCL [6] is another similar work based on the Frame concept, which captures design knowledge and represents variability. The XVCL approach prepares all possible combinations of design choices and corresponding code components beforehand. Then, a given feature model matches one combination and code components are selected easily. Rhizome avoids this code selector approach, since it is not scalable to prepare for all possibilities in advance when there are a large number of features. The template-based approach Rhizome uses instead embeds tags representing variability and code generation instructions. The code block to replace each marker is generated on-the-fly by the code generator. Feature Oriented Programming (FOP) and the AHEAD toolkit [7] is another work that inspired the Rhizome project. FOP is based on the concept of stepwise development where complex programs are built by incrementally adding features. First, features are described using Jak, an extension language to Java. Then, Jak feature files are aggregated and converted to Java code using a translation tool. The Jak language does not explicitly handle feature dependency and only supports Java code generation. Rhizome has better support in both aspects.

In the remainder of the paper we describe the feature modeling language and code generation approaches used by Rhizome, and the code generation unit (CGU), the core abstraction that binds feature models to code generation.

2. Feature Modeling Language

A feature model represents one of many possible ways to conceptually design a system. Based on how features are implemented in the source code, we divide them into two categories: data features and behavior features. A *data feature* generally corresponds to entities and properties in the source code and defines design choices to compose a hierarchical feature structure or property values. An example is a data feature called *user_type*, which has design choices of Teacher, Student, and Teaching Assistant. The Teacher feature can be further subdivided by considering it a sub-feature having its own tree structure that includes various properties. A *behavior feature* is associated with code that implements activities. For example, we may have a behavior feature called *teacher_activities*, which defines various activities associated with a Teacher user type, such as create, modify and delete students, compose questions and exams, grade and publish exam results, etc.

The feature modeling language, FeatureML, models data and behavior feature types and represents a feature model as an acyclic graph. For data features, FeatureML includes language elements for primitive features such as enumeration values, and scope values. It also contains referential elements for parent-child tree structures or superclass-subclass inheritance structures. For behavior features, FeatureML has language elements such as textual feature descriptions and dependencies to other data features or behavior features.

In addition to data and behavior features, FeatureML also contains an element called the *code_generation_unit* (CGU). This element bridges the gap between a feature model and template-based code generation. A code template usually has multiple tags and each tag is described using the template language called MarkerML to perform a particular code generation tasks using input parameters. A CGU element helps the code generator interpret feature design choices into parameter values and then use these parameter values in each tag in code templates.

```
<code_generation_unit unit_id="37"
        unit_name="UserLoginPage_java_tag_expansion"
        description="expand tags to create UserLoginPage
        class that represents the login page for the system"
        template_location="edu/ucsc/cse/exam/auth/
        UserLoginPagejava.tmpl">
   <tag_expansion tag_id="page_redirection">
      <parameter_content lookup_type="Function"
        parameter_name="userEntityName">
        findNonParentclassUserEntityNames
      </parameter_content>
   </tag_expansion>
</code_generation_unit>
```

Figure 1. Example of a CGU element.

Figure 1 shows an example of a CGU element. There are four CGU commands: file copy, snippet join, string replacement, and tag expansion. This example shows a tag expansion command, which specifies that the final code is generated by expanding a tag in the template file *UserLoginPage.java.tmpl*. This template file contains a tag identified by its tag_id *page_redirection* and this tag uses values for an input parameter *userEntityName*. Each parameter value is used to produce a specific code item based on the tag definition. For example, if there are 3 values for this *page_redirection* parameter, then there will be 3 code items generated, each using one parameter value. To get a list of values for this parameter, the code generator invokes an external function called *findNonParentclassUserEntityNames* during execution of the tag expansion command. There are other parameter value lookup methods such as directly providing string values, or using path expressions to query the in-memory parameter data structure. The tag expansion command corresponds to one of the code generation patterns—details on how the tag should be expanded are specified in the tag using the MarkerML template language. For example, one tag can generate a list of similar items, another tag can produce an if-else structure, and both tags belong to the tag expansion generation pattern.

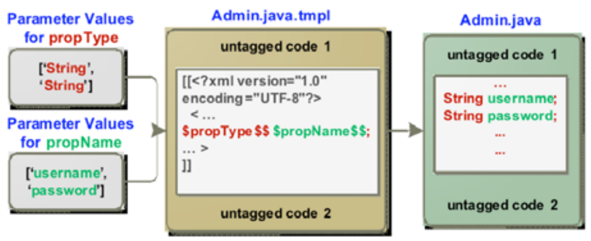

Figure 2. Concept overview for tag_expansion pattern.

Figure 2 illustrates how a tag expansion works. In a template file *Admin.java.tmpl*, there is a tag (enclosed between [[and]] characters) defined using MarkerML. This tag is used to generate a list of similar code items using a simple code structure for variable definition. This code structure contains two parameters *$propType$$* and *$propName$$*. Both parameters have 2 values, so these two parameters are replaced with corresponding values in each round and generate two list items in the final code.

A feature model contains a list of CGU elements and the code generator processes these CGUs one by one until all selected code templates are transformed into source code. A single CGU element may contain multiple code generation commands, and each CGU element may create multiple source code files. In general, the order of CGU processing is linear. But for some CGU types, such as snippet join, there can be precedence dependencies where a CGU must wait until its dependent CGUs finish processing first. We enumerate existing code generation patterns in the following list—each pattern corresponds to a CGU command.

- **Inter-file level generation patterns:**
    - **File_copy:** copy a template file to create a new file, without changing its contents.
    - **Snippet_join:** combine the contents of several files to create a new file.

- **Inner-file level generation patterns:**
  - **String_swap:** replace string_swap markers in the template file with string parameter values.
  - **Tag_expansion:** replace tags in the template file with generated code blocks. Tags are defined using the MarkerML language.

Currently, the MarkerML template language supports six types of recurring code structure patterns:

- **COPY:** simply copy the MarkerML template content into the generated code block without making changes.
- **LIST_GEN:** generate a list of similar items using a MarkerML template. Each list item is generated by filling in different parameter values.
- **LIST_GEN_CONDITIONAL:** use multiple MarkerML templates to generate a list of items. Each template is attached with a condition. Both conditions and templates contain parameters. During list item generation, each condition is checked with its parameter values filled in, and the first MarkerML template with a "true" condition is selected for list item generation.
- **LIST_GEN_BY_INDEX:** use multiple MarkerML templates to generate a list of items. Each template is associated with a range of indexes, and a specific MarkerML template is selected if the current parameter value falls into its associated range. For example, if a parameter has three values, we can specify that the first parameter value use one MarkerML template and the remaining two use another MarkerML template. This type is useful to generate "if, else if, else" code structures.
- **LIST_GEN_BY_COUNTER:** use a special *index* parameter in the MarkerML template that keeps track of the current parameter index. This is particularly useful for generating code related to index-based data traversal, e.g. traversing an array with each element's array index.
- **COUNT:** use a special *counter* parameter in the MarkerML template, useful for declaring index-based data structures such as arrays or lists. For example, when we define an array, we need to know its capacity upfront.

3. Code Generation

3.1. Code Generation Process

Code generation starts with an input document (a feature model encoded in FeatureML) that contains feature model design choices and CGU definitions. Most CGUs are processed in a linear order. CGUs with dependencies are processed based on their dependency structure—those at the leaf nodes are processed first before their dependent nodes can be processed. Design choices are interpreted as parameters and stored in an in-memory data structure. This data structure is a mixture of list and hash tables, which can be easily queried to retrieve parameter values using path expressions or customized function calls. Based on command types defined in CGUs and code generation patterns specified in tags using MarkerML, a CGU processing module fetches required template files, looks up parameter values, and performs corresponding generation work. Each CGU produces at least one source code file and some template files can be used multiple times to generate different code files. After all CGUs are processed by the code generator, code files that implement the input feature model are created. In the end, a finalize module is invoked to clean up the in-memory data structure and delete the in-memory DOM tree representing the feature model. Figure 3 shows an overview of the code generation process.

3.2. Code Generator Implementation

Rhizome's code generator uses a template-based generation approach. This approach does not require semantic knowledge about the implementation language and treats input as plain text only. As a result, the generator can produce virtually any text format as long as templates are available in the same format. Since most of the generation work is related to text processing, the code generator is implemented using Python due to its rich set of text processing libraries and flexible list

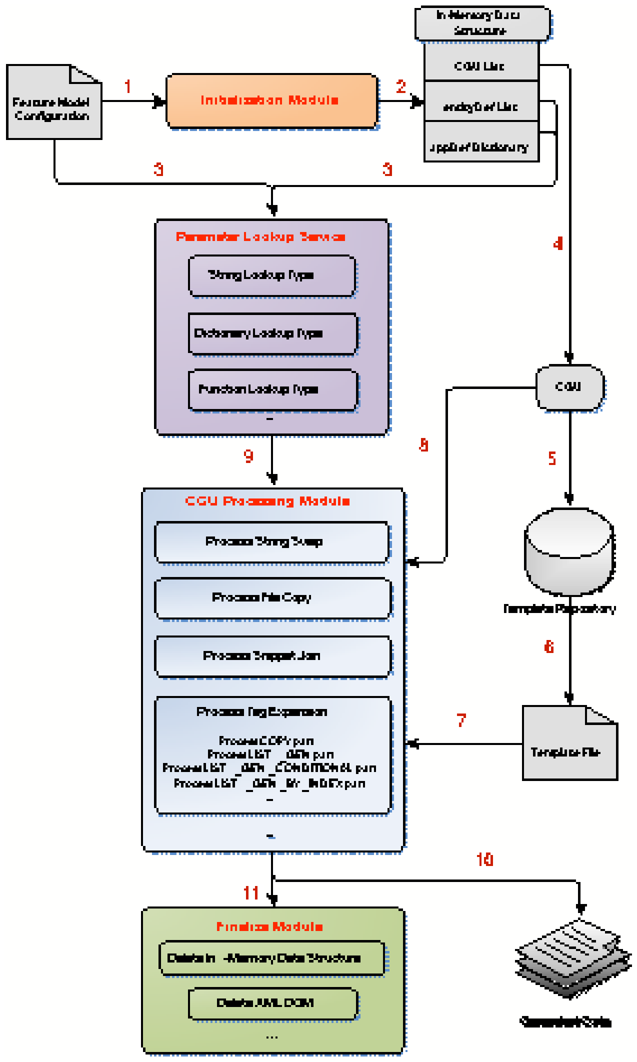

Figure 3. Code generation process overview.

and hash table (dictionary) data structures. The generator is around 2.2K lines of code and takes roughly 1 minute to produce source code for a feature model with 150 features (around 3.1K lines of XML code) on a laptop with a 1.6GHz Pentium CPU and 768MB memory running Python 2.5.2.

We applied Rhizome to the domain of online exam systems. An online exam system is a Web application that allows students to take online exams created by teachers. It has various user types including Teacher, Student, TA, and Administrator. Each user type has specific activities associated with it. For example, a Teacher can create/update/delete/list questions and exams, manage students of his class, grade exams and release exam reports, etc. A Student can take exams and view exam reports. A TA shares some responsibility with a Teacher, such as grading exams. An Administrator is the system manager and has access to all activities. Each user is authenticated using basic username/password pairs stored in repository and all activities are authorized based on user types.

We generated two variants of the online exam system. One has only one multiple choice question type, only Student and Teacher user types, and just basic activities. The other variant adds a true-or-false question type plus an additional administrator user type. These two generated systems have approximately 6K lines of code and used various application frameworks and libraries, such as the Apache Wicket Web application framework, Spring framework, Hibernate Object Relational Mapping, MySQL and Jetty Web application server. So, the generated systems are mid-size Web applications with system architectures representative of real world multi-tier Web applications. Rhizome appears to be useful for other application domains as well. A detailed demonstration can be viewed at [8].

4. Conclusion

We designed and implemented a feature modeling and automatic generation platform called Rhizome. Using this platform, software designers apply their high level design expertise to build feature models using the FeatureML modeling language. They simply pick feature design choices that have been implemented by platform developers. Platform developers are development experts that study a software product line to encapsulate similarity and variability in code templates. These templates have embedded markers at locations where variability needs to be resolved using feature model design choices. The code generator processes templates and replaces markers with generated code blocks using parameter values obtained from the feature model design choices.

Our approach is different from model-based code generation which only focuses on specific domains like GUI generation with graphical widgets. Our generator is also different from the code assembler approach where all possible design choice combinations are pre-calculated and code is prepared in advance. In Rhizome, platform developers have no information about possible design choices a software designer might choose. They only have information like “here, in the code, should be a property type and there should be a property name and a property value”. The actual property type, name and value are decided on the fly by designers when they build a feature model. Platform developers do not prepare templates based on all possible feature choice combinations, but based on each feature and let the code generator create code for these combinations.

This template-based code generation approach relies on template developers to provide insights and expertise for reusable code structure and architecture design. We feel this is a more flexible and general approach than using domain-specific modeling languages which only covers a specific domain. This generation method also makes code generator easier to implement and independent of programming languages or specific domain details.

Our work also identified several code generation patterns that can be reused for other code generator implementations. These generation patterns are discovered in real world applications to solve practical code generation problems. The pattern description language and processing module in the code generator can be easily reused in other code generation frameworks.

5. References

[1] G. Ge, "Rhizome: A Feature Modeling and Generation Platform for Software Product Lines," Department of Computer Science, Ph.D. Thesis, Santa Cruz: University of California, Santa Cruz, 2008.

[2] P. Clements, L. Northrop, and L. M. Northrop, Software Product Lines : Practices and Patterns, 3rd ed.: Addison-Wesley, Aug 20, 2001.

[3] K. Pohl, G. Böckle, and F. J. v. d. Linden, Software Product Line Engineering: Foundations, Principles and Techniques, 1st ed.: Springer, Sep 2005.

[4] Object Management Group, "Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification," July 2007.

[5] pure-systems GmbH, "Variant Management with pure::variants (Technical White Paper)."

[6] H. Zhang and S. Jarzabek, "XVCL: A Mechanism for Handling Variants in Software Product Lines," Science of Computer Programming, vol. 53, pp. 381-407, Dec 2004.

[7] D. Batory, "A Tutorial on Feature Oriented Programming and the AHEAD Tool Suite," in Generative and Transformational Techniques in Software Engineering, vol. 4143/2006, 2006, pp. 3-35.

[8] G. Ge, “Rhizome Demo,” www.soe.ucsc.edu/~guozheng/ Rhizome-demo-video/Rhizome-demo-new-March27-2008.html, last accessed: July 2, 2008.