An interpreter for framework-specific modeling languages

Kacper Bak, Steven She

Abstract

Framework-specific modeling languages (FSMLs) are specifications of framework concepts and their intended usages. Interpreting an FSML over a set of applications enables a user to detect framework concepts in source code through reverse-engineering.

We describe our FSML interpreter written using Prolog. The interpreter itself consists of two main components: a parser that is responsible for reading an FSML specification and an engine, responsible for interpreting the constraints of the FSML and reverse-engineering an FSML instance from a set of program facts. We discuss the purpose of each component and further highlight the advantages, as well as disadvantages of our implementation.

1. Introduction

Framework-specific modeling languages (FSMLs) are domain-specific languages designed to formalize framework concepts and their intended usage patterns [1]. For example, an FSML can specify that implementing a key listener in a Java applet involves implementing the KeyListener interface, registering the listener and finally de-registering the listener when the applet is closed. These usage patterns are currently not checked or detected by compilers. FSMLs augment the source programming language with domain-specific concepts and constraints enabling more sophisticated program checking and code assistance.

FSMLs are specified as a *cardinality-based feature model* [2]. The semantics of the features are defined through *mapping definitions*. A feature model consists of a feature tree and a set of additional cross-tree constraints (which can be empty). The feature tree specifies dependencies between features in a hierarchical manner such that children features can only be selected if their parent feature is selected. In addition, features have an associated cardinality. A feature can be optional $[0..1]$, mandatory $[1..1]$, have zero or more instances $[0..*]$ or any arbitrary upper and lower bound. Features can be also be grouped to specify an or-relationship. FSMLs have the concept of an *essential* feature. An essential feature is one that is necessary and sufficient for detecting the presence of its parent feature. Finally, mapping definitions describe the correspondence between features and code. They are similar to pointcuts in aspect-oriented programming [3].

```
applet ∗
    name ⟨fullyQualifiedName⟩
    extendsApplet ! ⟨assignableTo: 'Applet'⟩
        extendsJApplet ⟨assignableTo: 'JApplet'⟩
    listensToMouse?
        implementsMouseListener
            ⟨assignableTo: 'MouseListener'⟩
        registers ⟨calls: 'addMouseListener'⟩
        deregisters ⟨calls: 'removeMouseListener'⟩
    showsStatus ∗ ⟨calls: showStatus(String)⟩
        message : string ⟨argVal: 1⟩
```

Figure 1. Subset of the Applet FSML

In Figure 1, a subset of the Applet FSML is shown in our FML textual syntax. The names in bold text represent features. The cardinality of a feature is specified using syntax similar to regular expressions: a '$*$' denotes $[0..*]$, a '?' denotes $[0..1]$ and no modifier denotes $[1..1]$. Next, the angled brackets are used to specify a feature's associated *mapping definition*. Children of a feature are specified through indentation. The same rule applies to feature groups, but each group starts with cardinality specification in angle brackets (not shown in this example). An exclamation mark is used to denote an essential feature.

The constraints specified in a cardinality-based feature model can be translated to first-order logic. This translation enables reasoning and provides a means for validating whether a framework application satisfies the constraints imposed by an FSML.

A key concept of FSMLs is that the model is a derived artifact. A framework-specific model (FSM) is constructed by applying static analysis to reverse-engineer a model from the code of a framework application. This operation is where mapping definitions are used. Mapping definitions specify the criteria that need to be satisfied for a feature to be detected in code.

Unfortunately, statically determining the run-time semantics of a program is undecidable. Thus, we use *code queries* to approximate the detection of behavioural patterns. Interpreting a code query returns values in the source code that satisfy the query. Interpreting all code queries in an FSML returns all framework-specific features present in code.

In this paper, we present our prototype implementation for a FSML interpreter built using SWI-Prolog [4]. Our interpreter is capable of parsing an FSML specification, extract facts from a set of Java applications, reverse-engineer an FSM and check hierarchy and cardinality constraints. We describe the function of each component in our prototype, and where applicable, give a detailed description of its implementation. We provide a guide for running the tool in Section 6, describe future work in Section 7 and conclude in Section 8.

2. FML Parser

We have designed a textual grammar called FML for specifying FSMLs. FML is our domain-specific language for feature modeling enriched with cross-tree constraints and optional mapping definitions. It is under active development and here we present the first version of it.

FML is designed to model hierarchies of features with cardinalities and optional attributes. The scope of FML is much larger than what is presented here. We aim to develop FML into a language suitable for large scale feature modeling supporting thousands of features. For example, we would like model the Linux configuration system containing over 5000 features using FML [5]. There is a notion of single inheritance available in the language, but we are not clear about its semantics and thus do not use this feature in this project. Similar to many programming languages, it is possible to limit the scope of visibility by introducing local definitions. FML also supports the concept of feature reuse, which is similar to a function call.

The language offers a wide range of cross-tree constraints. Sample constructs include logical operators, *If-Then-Else*, arithmetic and set operations. Sets can be constructed either implicitly (by choosing elements satisfying some condition) or explicitly (by listing elements). Furthermore, constraints can also be specified by *Alloy-like* quantifiers [6], which determine that some condition is satisfied by all, none, some elements. As feature models are usually represented as trees, FML allows navigation within its hierarchical structure by means of dot (.) and hat (^) operators. The former goes one level down, while the latter goes one level up.

The current version of FML has quite well defined syntax, but is still unclear on semantics of some constructions. In addition, it uses curly braces for denoting features instead of whitespace rule. We think it would be convenient to offer both mechanisms for structuring code.

2.1. Implementation

The parser is built in Prolog as a set of *Definite-Clause Grammar* (DCG) rules [7]. Technically, it is a recursive descent parser that uses difference lists and Prolog backtracking. Difference lists represent the difference between the input and output stream. There is a straightforward correspondence between language grammar and DCG rules. Therefore, the default syntax-tree structure is very close to an input grammar.

DCGs are very easy to implement in Prolog and easy to use. As they are ordinary Prolog code, the output syntax tree can be processed during the parsing stage. A DCG specification in Prolog appears very similar to a grammar specification in standard Backus-Naur Form. SWI-Prolog provides convenient syntax for writing DCGs in the form of the `-->` operator. The `-->` operator is preprocessed by the SWI-Prolog compiler to automatically add arguments representing the difference list. Prolog backtracking is used to implement the parser's lookahead mechanism.

On the other hand, since backtracking is the main control flow mechanism, there is typically very poor performance in the case of complicated grammars. Backtracking also makes it harder to detect ambiguities in the grammar.

Our recursive descent parser makes substantial use of higher-order programming and is written in a rather concise style. However, it does not produce any error messages if the input is wrong; it simply fails instead. Parser's source code can be found in parser.pl file. The top-level parsing term is declared as *parse(File, AST)*, where *File* is a path to FML source code and *AST* is a resulting abstract syntax tree. A sample FSML in FML syntax is provided in the applet.fml file.

FML combines feature models with mapping definitions. Therefore, mapping definitions are part of input files. Mappings strongly depend on language of source programs. For instance, mappings for Java programs are completely different from XML mappings. For this reason, they should be parsed by specialized tools. Although a proper implementation will invoke other parsers for interpreting the mapping definition, we decided to embed some Java-specific mapping definitions

into the current FML implementation.

The Prolog implementation of the parser is certainly adequate for our prototype. Nonetheless, it is unlikely we will continue using Prolog for the parser component when building a more robust, real-world implementation.

3. Java Fact Extractor

A front-end for the Eclipse Java Development Tools (JDT) compiler [8] was written to construct a Prolog database from a set of Java applications. The database is used to reverse-engineer a set of framework-specific model in subsequent steps of our prototype. The extractor builds facts from three constructs of the Java language: class declarations, method declarations and method calls. We refer to the set of input source files as a *project*. Table 1 shows the facts and their arguments that are constructed by the extractor.

Class Declarations. Each file in the project corresponds to a class or interface declaration. We ignore interface declarations since they do not contribute any run-time semantics. We construct a **class** fact for each class declaration. A class fact contains its fully-qualified name (ie. its package and class name), its superclass, any implemented interfaces followed by a list of declared methods.

Method Declarations. For method declarations, we construct a fact **method** that contains its name, return type, list of parameter types followed by a list of method calls. A method's name is its signature, which consists of its declaring class followed by its method name and parameters.

Method Calls. Method calls are represented by the **methodCall** fact. They are nested under method declarations and contain three arguments. The first, is its receiving object name. We currently use this field to distinguish between self-references using this and known static fields such as System.out. The second argument is the signature of the called method followed by a list of resolved argument expressions. The argument resolution is based on some form of pointer analysis. We currently perform no pointer analysis and as a result, only constants are retrieved for arguments. If the argument cannot be resolved, a special none atom is returned.

Table 1. Extracted Prolog fact structure

Functor	Arguments
project	[classes]
class	Name, superclass, [interfaces], [methods]
method	Name, return type, [parameter types], [calls]
methodCall	Receiver object, signature, [arguments]

```
public class Simple extends Applet
                    implements MouseListener {
    public void init () {
        showStatus("foo");
        addMouseListener(this);
    }
    public void destroy() {
        showStatus("bar");
    }
    ...
}
```

Figure 2. Java source of a Simple applet

```
project([
class('Simple', 'Applet', ['MouseListener'],
  [method('init', 'void', [],
    [methodCall('this', 'showStatus(String)', ['foo']),
     methodCall('this', 'addMouseListener(...)',
       ['this']),
    ])
   ... -- destroy method omitted
  ])])
```

Figure 3. Prolog fact extracted from Java source

In Figure 2, the Java implementation of a simple applet is shown. The implemented methods of the MouseListener are omitted. This applet makes two calls to the showStatus method and registers itself as a mouse listener through the addMouseListener method call. Note that a call to removeMouseListener is absent. The Java fact extractor builds the Prolog fact shown in Figure 3.

3.1. Implementation

The extractor is built on top of the JDT compiler which is implemented in Java. JDT parses and compiles an AST for a source file and exposes it using the visitor design pattern. We have opted to use the Scala programming language [9] to implement the extractor. Scala is a hybrid language combining functional and object-oriented programming concepts that runs on the JVM and is inter-compatible with Java.

The internal representation of an AST in the JDT compiler is optimized for speed and space. Consequently, much of its structure is stored as primitives such as char arrays instead of strings for class names

and where possible, fields are used over methods.

Scala provides several benefits for interacting with such mixed-data structures. First, fields and parameterless method accesses are the same (ie. the empty parentheses following a method call can be omitted). Scala also provides a uniform way of interacting with Java primitives through its type system. In Scala, every value is represented as an object, thus, Scala does away with Java primitives. For example, an character array is written as Array[Char], which specifies an Array parameterized by the Char type. Conversions between Java primitives and Scala objects are performed automatically by the Scala compiler through *implicit* conversions. A further consequence of this type system is that Scala implements all operations as method calls; there are no reserved operators. For example, the expression 1 + 2 is in fact shorthand for the expression 1.+(2). As a result, Scala provides a convenient and uniform way of working with collections. For example, the higher-order functions map and append (using the ++ operator) are available to both Arrays and Lists. In fact, these operations are available to any type implementing the Iterable interface.

The fact extractor constructs an in-memory representation of the Prolog fact structure. We used a functional pretty-printer combinator library based on the paper by Wadler [10] to render the fact structure to text. The pretty-printer library provides combinators to concatenate, group, nest and line-break text fragments called documents. A call to a document's format renders the combined document as text that tries to respect a given margin width. These combinators are part of the Scala standard library.

4. Linearizing Facts

The tree structure generated by the Java fact extractor is a concise representation of the nesting relationship between the various constructs. However, this tree representation is difficult to query using Prolog. Therefore, we perform a step where we expand, or *linearize* this tree structure into a list of facts. The tree structure is retained through a unique identifier for each fact and an argument referencing its parent. In Table 2, a subset of the facts generated through this linearization is shown.

We use a single functor, creatively called **fact**, to store each linearized fact. A **fact** has five parameters: (1) is its type, (2) is the node id of its parent, (3) is the type of its attribute, (4) is the fact's own node id, and (5) is the attribute value.

This stage acts much like the parse function in our class interpreters. However, in this case, we convert the input tree structure into a flat Prolog-centric list of facts. These facts are then asserted into the Prolog database.

5. Engine

The engine is the central point of our project, as it glues together the two other components: the parser and extractor. The engine is responsible for instantiating framework-specific models by detecting the presence of features in source programs. Feature instantiation is done by invoking *code queries* which are approximations of the mapping definitions specified in the FSML. A code query returns all source code facts matching its criteria. For example, a code query for fullyQualifiedName would return all facts with a class' fully qualified name. The semantics of mapping definitions, queries and facts is separated from semantics of feature models since mappings are language dependent.

The engine provides a function *instantiate* that interprets an FSML's AST. To instantiate a feature, all subsequent parent features must be present. The presence of a feature is determined by whether its code query returns a result. If there are multiple results, then a feature is instantiated multiple times. A further check on a feature's essential subfeatures is also performed. If all essential subfeatures are present, then we instantiate the feature. At this stage, we perform a check of cardinalities. If a cardinality constraint is not essential but is still violated, the subfeature will be marked

Table 2. Linearized facts

fact type	parent	attr. type	id	attribute
project	–	jproject	n_1	
class	n_1	jclass	n_3	
assignableTo	n_3	string	n_4	'Applet'
name	n_3	string	n_5	'Simple'
implements	n_3	string	n_6	'MouseListener'
hasMethod	n_3	jmethod	n_7	'init'
returnType	n_7	jtype	n_9	void
calls	n_7	jmethod	n_{10}	'showStatus(String)'
recObject	n_{10}	string	n_{11}	this
argVal	n_{10}	tuple	n_{13}	(1, foo)
calls	n_7	jmethod	n_{14}	'addMouseListener...'
recObject	n_{14}	string	n_{15}	this
argVal	n_{14}	tuple	n_{16}	(1, this)
...remaining facts omitted.				

Represented in Prolog as:

fact(fact type, parent, attr. type, id, attribute)

Applet

name = Simple	showsStatus
extendsApplet	message = foo
listensToMouse	showsStatus
implementsMouseListener	message = bar
registers	

Figure 4. Framework-specific model

query(assignableTo, 'Applet', X).

X = **fact**(assignableTo, n_3, string, n_4, 'Applet').

query(calls, 'showStatus(String)', X).

X = **fact**(calls, n_7, jmethod, n_{10}, 'showStatus(...)') ;
X = **fact**(calls, n_7, jmethod, n_{18}, 'showStatus(...)').

Figure 5. Sample queries with results

by a *missing* tag. The tree is currently interpreted in depth-first search manner, but we may find other, more suitable strategies in the future.

The result from running *instantiate* with the asserted linearized facts is shown in Figure 4. Here, the framework-specific model is shown as a list of instantiated features and their associated attribute values if any. In this example, most features of the Applet FSML are instantiated. However, the deregisters feature is missing since it was not in code and thus, the code query for this feature failed to return a result. We will go into further detail on the implementation of code queries in the following sub-section.

After instantiation, the constructed model is passed to the constraint checker. Although cross-tree constraints can be as essential as constraints imposed by cardinalities, they are validated at later stage. The two functions are separated because additional constraints may refer to features which are not instantiated at feature model traversal time.

5.1. Implementation

Similar to other components, we implemented the engine and code queries in Prolog. Source programs are represented as asserted facts in Prolog database. These facts make up a linearized tree structure which resembles the Java source AST. This representation makes code queries very simple, concise and extensible.

All code queries operate on a single fact, and thus have very similar structure. That makes metaprogramming in Prolog an easy task. Although queries are defined statically, each query call is constructed dynamically when interpreting the FSML tree by using the *univ* (=..) operator. The univ operator transforms a list of elements into a term where the head of the list acts as the functor and the remaining elements act as arguments. In our case, the functor is always the **query** term. The first argument to a query is the fact type (ie. assignableTo or calls) and determines the semantics of the query. The next arguments of the query term represent input parameters of the query. For example, the assignableTo query has a parameter for specifying the class name. Finally, the last argument is the resulting fact. The constructed **query** term is executed as a Prolog goal and then returned if the goal can be satisfied, otherwise it fails. This metaprogramming approach lets us use Prolog's searching capabilities and also makes code very reusable.

In our running example there are two sample mapping definitions — assignableTo: 'Applet' and calls: showStatus(String). They are translated into two queries, each with one argument. The first query returns facts that are superclasses of Applet. The second query searches for facts about a method call. Here we are interested in finding calls to the showStatus method with String parameter. Figure 5 shows how the queries are invoked and what results they return. The fields with n_i uniquely identify each node and are used to reference their parent nodes.

The *instantiate* term traverses FSML abstract syntax tree in depth first search fashion and instantiates features by executing relevant code queries. There is direct correspondence between mapping definitions in FSML AST and available code queries. Although they should be separated from the interpreter, we decided to combine the two artifacts for this project. Each code query returns only a single fact from the database, but retains additional results through backtracking. Therefore we use the **findall** predicate to fetch all relevant facts. Facts are matched not only on basis of mapping definitions, but also by parents. Matching by parents is the only way to associate a particular feature instance with instances of subfeatures. For example, if there are two applets, and only one of them implements MouseListener interface, then we want to associate this fact with proper applet instance. After query execution, the number of returned facts is checked with its cardinality constraint from the model.

Information on whether a subfeature is correct or not bubbles from leaves up to the FSML root. Our implementation of error propagation is not very far from the *Maybe Monad* known from functional programming [11]. This idea will be pushed further in future implementations.

The engine still has very limited capabilities. First of all, it is not complete, which means that it can interpret only the most basic constructions of FML. We are still unsure about semantics of certain constructions. Besides, FML is rather non-trivial to interpret if we take into account possible interactions between modules and cross-tree constraints. Another limitation is the analysis of only a subset of the constructs available in code. The engine also has performance issues related with the way Prolog searches its database. We tested the interpreter on a project containing 20 applets and about 11 thousands facts. Instantiation of our applet models took about 90 seconds on a modern computer (Intel Core2 Duo, 2.40GHz).

6. Operation

The java fact extractor is available as a compiled jar containing all dependencies[1]. The extractor is executed with the following command:

java -jar extractor.jar ⟨-f output.pl⟩ ⟨source files | directories⟩

The first option, -f, specifies whether the output is sent to a file instead of standard output. The remaining parameters specify any source files or directories for the extractor to analyze. Any additional options are passed to the internal JDT compiler.

We provide two extracted Prolog projects with our tool. The first, simple.pl is an example similar to the one presented in this paper. A larger project, sun.pl consists of the extracted facts for 20 Java applet downloaded from the Java applet tutorials available from Sun.

Model instantiation starts with loading engine.pl file into Prolog interpreter. Assuming that the Prolog interpreter is started in the project directory, it is enough to type [engine]. Next, it is required to load the extracted AST of the source project. For example, this could be either [simple] or [sun]. Finally, execute run(*File*), where *File* is path to the FML source file. We provide the example Applet FSML shown in this paper as applet.fml. The run term parses the input feature model, extracts facts about the source project, instantiates the framework-specific model and prints the output model.

7. Future Work

Our project is still in a very early stage of development. First of all, we are going to work on both syntax and semantics of FML. There are still many questions about useful features of the language. We would like to determine which constructions are just syntactic sugar and which are beyond the expressiveness of pure feature models. Another work concerns code mappings and their placement within feature models. It is still not clear whether they should be mixed with feature models or defined separately. We would also like to work on preprocessing the FSML AST returned by parser. Its current form is very similar to the parse tree and complicates interpretation and the engine itself.

The extractor component extracts only basic facts from the source program. Our current prototype does not implement any form of pointer analysis. For example, the introduction of a variable is enough to cause the analysis to fail at detecting the value of an method call argument.

There is already published work on using declarative languages for specifying and executing static pointer analysis. Bravenboer presented work on using Datalog to specify points-to analysis [12]. Datalog is a syntactic subset of Prolog that guarantees that query evaluation is sound and complete. We can apply similar techniques to implement the static analysis necessary for executing more complex code mappings (e.g. control-flow and data-flow analysis).

Our interpreter relies on an extractor to build a preloaded set of facts. Realistic programs can contain a large number of facts, therefore we will have to determine which information should be stored in the database and which to gather at run-time. Incorporating run-time analysis would mean connecting our Prolog implementation with a JVM-based language. There are tools for accomplishing this task such as the JPL project from SWI-Prolog [13].

The engine component implements only a very limited subset of FSML syntax tree. It still requires more design and development work, which is tightly coupled with the parser part. Some of missing and non-trivial features include support for modules and navigation capabilities. We also plan to improve code by making it more higher-order and fully apply *Maybe Monad* concept. Furthermore, the abstract syntax tree with resulting FSM should be simplified by unifying nodes representation.

The *instantiate* term currently performs only basic cardinality checks; it does not validate additional constraints. Certainly, *check* should be a separate stage, but we are still considering using specialized tools, such as Alloy, for this task.

Finally, type checking can be used to validate the composition of mapping definitions. Mapping definitions can be composed through the parent-child relation in the tree hierarchy or through cross-tree constraints. For example, a common pattern is to nest

1. http://www.eng.uwaterloo.ca/~shshe/extractor.html

the argVal mapping under a call mapping. However, it is possible to nest semantically invalid mappings, such placing an argVal as a child of a class. We currently make no attempt at detecting these semantically invalid mappings. This is an exciting task and we hope to be able to address this in the near future.

8. Conclusions

In this project, we have developed an interpreter that parses an FSML specification and creates instances of the FSML through Prolog queries. Furthermore, we have implemented a fact extractor for Java programs using Scala. In this paper, we identified the main components of the infrastructure and interactions between them. Our prototype implementation is capable of supporting the reverse-engineering of simple FSMLs. Although there are still many challenges to overcome, we believe that Prolog has, and will continue to be an effective language for implementing our interpreter.

References

[1] M. Antkiewicz, "Framework-specific modeling languages," Ph.D. dissertation, University of Waterloo, Sep 2008. [Online]. Available: http://hdl.handle.net/10012/4030

[2] K. Czarnecki, S. Helsen, and U. Eisenecker, "Formalizing cardinality-based feature models and their specialization," in *Software Process: Improvement and Practice*, 2005, p. 2005.

[3] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *ECOOP*, 1997, pp. 220–242.

[4] J. Wielemaker, "Logic programming for knowledge-intensive interactive applications," Ph.D. dissertation, University of Amsterdam, 2009, http://dare.uva.nl/en/record/300739.

[5] S. She, R. Lotufo, T. Berger, A. Wąsowski, and K. Czarnecki, "The variability model of the linux kernel," in *VaMoS*, 2010, submitted for review.

[6] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[7] C. M. Sperberg-McQueen, "A brief introduction to definite clause grammars and definite clause translation grammars," W3C, 2004. [Online]. Available: http://cmsmcq.com/2004/lgintro.html

[8] Eclipse Foundation, "Eclipse java development tools (JDT)," 2009. [Online]. Available: http://www.eclipse.org/jdt/

[9] "The Scala programming language," École Polytechnique Fédérale de Lausanne (EPFL), 2009. [Online]. Available: http://www.scala-lang.org

[10] P. Wadler, "A prettier printer," in *Journal of Functional Programming*. Palgrave Macmillan, 1998, pp. 223–244.

[11] J. Newbern, "The maybe monad." [Online]. Available: http://www.haskell.org/all_about_monads/html/maybemonad.html

[12] M. Bravenboer and Y. Smaragdakis, "Strictly declarative specification of sophisticated points-to analyses," in *OOPSLA*, S. Arora and G. T. Leavens, Eds. ACM, 2009, pp. 243–262.

[13] P. Singleton, "A SWI-Prolog to Java interface," 2009. [Online]. Available: http://www.swi-prolog.org/packages/jpl/prolog_api/overview.html