Automatic Extraction of Framework-Specific Models from Framework-Based Application Code

Michał Antkiewicz, Thiago Tonelli Bartolomei, Krzysztof Czarnecki
University of Waterloo
200 University Ave. West
Waterloo, ON, Canada
{mantkiew, ttonelli, kczarnec}@uwaterloo.ca

ABSTRACT

Framework-specific models represent the design of application code from the framework viewpoint by showing how framework-provided concepts are implemented in the code. In this paper, we describe an experimental study of the static analyses necessary to automatically retrieve such models from application code. We reverse engineer a number of applications based on three open-source frameworks and evaluate the quality of the retrieved models. The models are expressed using *framework-specific modeling languages* (FSMLs), each designed for one of the open-source frameworks. For reverse engineering, we use prototype implementations of the three FSMLs. Our results show that for the considered frameworks rather simple code analyses are sufficient for automatically retrieving framework-specific models form a large body of application code with high precision and recall.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.13 [**Software Engineering**]: Reusable Software—*Reuse models*

General Terms

Design, Documentation, Languages

Keywords

reverse engineering, framework-specific models, framework-specific modeling languages, static analysis, object-oriented frameworks

1. INTRODUCTION

Object-oriented frameworks are widely used to implement reusable designs. Domain concepts provided by a framework are implemented in the application code by writing *framework completion code*. Unfortunately, the concept instances are often not easily recognizable by developers directly in the code because they are scattered across and tangled with the code and buried in a large amount of implementation details. *Framework-specific models* have been proposed to address this problem by offering an abstract view of the application code from the viewpoint of the framework [2, 3]. Such models explicitly represent the instances of framework-provided concepts that are implemented in the application code.

One way of formalizing framework-provided concepts is by decomposing them into hierarchies of *features* [3, 8]. Features are distinguishing characteristics/properties of a concept and allow discriminating among concept instances. Features may correspond to *structural* or *behavioral patterns* in the application code. Identifying the matches of the patterns in the code allows determining the presence and the values of the features in the model. While the structural patterns can be determined statically with full precision by simple code queries, precisely determining matches of behavioral patterns in the application code using static analysis is potentially undecidable.

In this paper, we report on a study we conducted to measure the precision and recall of reverse engineering using code queries that locate matches of behavioral patterns in the completion code. The study was executed in two phases: i) identification of types of patterns and their corresponding code queries, and ii) evaluation of the precision and recall of code queries. In the first phase, we analyzed mappings attached to features from three example Framework-Specific Modeling Languages (FSMLs) [3] designed for Java Applet [20], Apache Struts [5], and a part of the Eclipse Workbench [18] frameworks. FSMLs are a kind of domain-specific modeling languages that are designed for an area of concern of an object-oriented framework and can be used for expressing framework-specific models. The result of the analysis is a classification of patterns that the features correspond to and code queries that were implemented in the prototypes of the FSMLs. In the second phase we reverse engineered a large number of example applications built on top of the three frameworks using the prototype implementations of the FSMLs. We then verified the correctness of the retrieved framework-specific models and calculated precision and recall of the code queries. Additionally, we categorized common false positives and false negatives of the code queries and proposed refined versions that would reach 100% precision and recall for the studied applications.

The intended audience of this paper is primarily tool builders interested in retrieving framework-specific models from

ASE'07, November 5–9, 2007, Atlanta, Georgia, USA.

application code. They can learn about specific approximations of behavioral patterns and their effectiveness. Also, the reverse engineering community interested in analysis of framework completion code can learn about the particulars of the analysis in the context of frameworks. Another target audience is researchers working on static analyses. These researchers can use the study results as a source of ideas of how the retrieval of framework-specific models could benefit from improved static analyses. And finally, the designers of modeling languages can learn about different types of structural and behavioral code patterns that can be used to define the mapping between models and code.

The main contribution of this paper is providing evidence that the efficient and precise retrieval of models that represent the dynamic interaction between application code and frameworks is feasible. We argue that by concentrating on the framework boundary and leveraging framework-specific knowledge, rather simple code queries become sufficient. Furthermore, we provide precise definitions of behavioral code patterns using meta pointcuts, code queries that can be used for pattern retrieval approximations, and discuss possible false positives and negatives of those queries.

The remainder of the paper is organized as follows. In Section 2 we motivate the study and discuss the challenges of and the requirements for static analysis of framework completion code. In Section 3 we provide the necessary background information on framework-specific models. Next, in Section 4 we describe the setup of the study we conducted, followed by Sections 5 and 6, where we describe the resulting data. In Section 7 we discuss the data as well as the threats to validity. We present the related work in Section 8 and conclude the paper in Section 9.

2. CHALLENGES OF STATICALLY ANALYZING COMPLETION CODE

Framework-specific models describe how concepts provided by the framework are implemented in the completion code. Concept instances are characterized by configurations of features and the features correspond to structural and behavioral patterns in the completion code. Therefore, automatic extraction of framework-specific models requires the retrieval of structural and behavioral patterns from the completion code using static analysis.

Unfortunately, static analysis of framework completion code is difficult. One reason is the *inversion of control* inherent to framework design, whereby the main threads of control belong to the framework and the framework passes the control to the application using *callback methods*. Due to the inversion of control, both the application and the framework need to be considered during the analysis. Also, frameworks commonly interpret configuration files and use reflection to dynamically load and instantiate application classes. Therefore, the construction of the complete and precise control flow graph, which is a basis for many other static analyses, is often impossible.

However, we believe that analyzing the complete code of both the application and the framework is not necessary. To understand how an application is using a framework and extract framework-specific models, one must focus on the *framework boundary*, that is, all places in the code where the application interacts with the framework. The framework boundary consists of all callback methods implemented in the application and all references to the framework code from the application.

Another characteristic of framework-based code is the use of configuration files, which are declarative specifications interpreted by a framework. The configuration files are used not only for specifying parameters to the framework, but also for assigning roles to code elements, such as classes and methods, and defining relationships among code elements. In some cases static analysis of the completion code is not possible without interpreting the configuration files because code elements are indistinguishable when only considering the code (e.g., in Spring[1] and Java Server Faces[2] frameworks).

Therefore, the retrieval of framework-specific models requires both utilizing configuration files and using code queries that i) do not require complete control flow graph information and ii) perform the required static analyses on-demand (i.e., compute partial control flow graphs).

Given these challenges and requirements for static analysis of the completion code, we propose a number of code queries that can be used for framework-specific model extraction. The proposed code queries are both *incomplete* and *unsound* approximations of behavioral patterns. In the next section we describe how framework-specific models correspond to code patterns and how the correspondence can be defined.

3. FRAMEWORK-SPECIFIC MODELS

Framework-specific models describe framework-provided concepts as implemented in application code. For example, consider a web application based on the Apache Struts framework. The framework provides concepts such as *form*, *action*, and *forward*. Forms accept input from the users and actions process submitted forms. Actions return forwards, which link to other actions or web pages. Instances of these concepts may include a *user login* form and a *login* action, which can return *success* and *access denied* forwards. A framework-specific model for our example Struts application would include these concept instances and could be used, for example, to visualize the page flow of the application.

Frameworks impose sets of requirements that the completion code must satisfy in order to instantiate a certain concept. Since such requirements can be fulfilled in many different ways, instances of framework-provided concepts are usually not uniform. For example, the Struts framework prescribes subclassing and appropriate declarations in an XML configuration file for the implementation of the concept *action*. Also, actions can be implemented as, among other choices, a basic action, a dispatch action, or a forwarding action. In each case, a different framework-provided class needs to be subclassed and values of different sets of attributes need to be used in XML action declarations.

Framework-provided concepts can be captured in framework-specific modeling languages (FSMLs) [3]. The abstract syntax of an FSML defines a decomposition of a concept into a hierarchy of *features*. Features represent distinguishing characteristics of concepts and can be used to discriminate among concept instances. Consequently, concept instances are described by *configurations* of features. In the feature hierarchy, features can be *essential*, *mandatory*, and *optional* with respect to their parent feature. In a feature configu-

[1] `http://www.springframework.org/`
[2] `http://java.sun.com/javaee/javaserverfaces/`

ration, if the parent feature is present, an essential feature *must* be present, a mandatory feature *should* be present and an optional feature *may* be present. A parent feature cannot be present without its essential subfeatures, whereas missing mandatory subfeatures indicate an error in the configuration of the parent feature. A feature may also have a *type* meaning that a value of that type can be associated with the feature in the configuration. The columns labelled *FSML feature* of Tables 3-5 contain features of the concepts we considered in the study. The hierarchy of features is represented using indentation (subfeatures are further right). Cardinalities of features are indicated in square brackets: [0..1] for optional features, [1] for mandatory features, and [0..*] and [1..*] for multiple features. Essential features are marked using exclamation mark (!). The abstract syntax tree of a framework-specific model is a feature configuration and closely resembles the feature hierarchy. Such a model contains instances of selected features, possibly many instances of multiple features, and values associated with features with types.

Features describing a concept instance can correspond to structural and behavioral patterns in the completion code that implements the instance. For example, the essential feature of an action specifies that the action must be assignable to Struts' `Action` class. Other features of an action correspond to the method `execute` or *action methods* if an action is a dispatch action. Yet other features correspond to method calls used to find forwards returned by action methods. Features can also correspond to action and forward declarations in an XML configuration file and to XML attributes of these declarations, such as *forward name*. The correspondence between the features and the code is specified in the *mapping of the abstract syntax to the framework API* of an FSML [3]. A feature can be associated with a *mapping definition*, which can be realized by a *code query* and a *code transformation*. A code query matches a pattern in the completion code. A code transformation creates, updates, and removes a pattern corresponding to a feature in the completion code. For example, instances of the concept *action* could be determined by a code query returning classes assignable to the Java class *Action* (based on the essential feature). A code transformation can create or remove Java classes implementing instances of the concept *action*. The mapping enables automated round-trip engineering, where the code can be created from the model, the model from the code, and changes made to the code and the model can be identified and reconciled [3].

In this paper we focus on the identification of the types of code patterns that feature instances can correspond to, and on the evaluation of code queries that realize the mapping between the features and the completion code.

4. SETUP OF THE STUDY

We conducted the study in two phases. The purpose of the first phase was to identify types of structural and behavioral patterns that need to be matched in the completion code in order to retrieve framework-specific models using the three FSMLs. The purpose of the second phase was twofold: i) determine the precision and recall of the code queries used in the prototypes for the location of code patterns, and ii) propose refined versions of the code queries that would provide 100% precision and recall.

The input to the first phase of the study are three example FSMLs, one for each of the following frameworks: Java Applet [20], Apache Struts [5], and part of Eclipse Workbench [18] (detailed descriptions of the languages can be found elsewhere [3]). The metamodels of the FSMLs consist of abstract syntax and mapping definitions. Applet FSML captures the concept of Java *applet* and has 20 features. Struts FSML captures the concepts of *action*, *form*, and *forward*, and has 43 features. It addresses the problem of maintaining the referential integrity between Java code and XML configuration file. Eclipse Workbench Part Interaction (WPI) FSML captures the concepts of *editor*, *view*, *selection provider*, *selection listener*, *part listener*, and *adapter provider/adapter requestor*. WPI FSML has 52 features and models the interactions that can potentially occur among workbench parts. WPI FSML also encodes many framework rules and helps with maintaining the referential integrity between Java code and XML plug-in manifest files related to part ids.

In this study we considered only features related to Java code and omitted features related to XML configuration files because such features can be retrieved with 100% precision and recall as well as features that represent referential integrity constraints, which are realized by model queries. The identified types of code patterns and the implemented queries are presented in Section 5.

In the second phase of the study we used the prototype implementations of the three FSMLs [3] to automatically reverse engineer a number of example applications. The prototypes implement code queries that realize mapping definitions of the FSMLs. The unit of analysis was a *project*: an abstract entity that groups all source artifacts of the analyzed application. For the Java Applet framework, example applets were grouped into two projects, one with 20 examples provided by Sun and one with 51 applets collected from the Internet. Each of the Struts applications, Apache Roller [4] (v.3.0), Mailreader [5] (v.1.3.8), and Cookbook [5] (v.1.3.8), constitutes a separate project. For the Eclipse Workbench framework, an application is encapsulated as an Eclipse plug-in. Because Eclipse plug-ins form complex dependency graphs, it is not possible to analyze plug-ins separately: common plug-in dependencies (such as *org.eclipse.ui*) would be analyzed multiple times. However, by analyzing one of the most-dependent plug-ins, we can analyze that plug-in and all other plug-ins it depends on at once. In this study, the project consisted of the *org.eclipse.pde.ui* plug-in (v.3.2), which depends on many other ui plug-ins including[3] *ant.ui*, *debug.ui*, *jdt.debug.ui*, *jdt.ui*, *ui*, *ui.editors*, *ui.ide*, *ui.views*, and *ui.workbench.texteditor*. This allowed us to analyze part interactions that can occur among 88 workbench parts (editors and views).

The result of the analysis reveals the precision and recall with which the queries were able to approximate the code patterns. By manually inspecting the code we were able to identify categories of patterns missed by the used code queries. Subsequently, we proposed refined versions of code queries that would capture the patterns missed by the original code queries. The queries obtained from this iterative process are presented in Section 5 and the data relative to their precision and recall is described in Section 6. All results are discussed in Section 7.

Data collection process. For a feature *f*, we consider

[3]We omit prefix *org.eclipse.* from the names.

Structural Pattern Expr.	Structural Element(s) Matched	Abbreviation
c assignableTo: t	matches if objects of the class c are assignable to the type t	assignable
f fieldOfType: t	matches if objects of the type t are assignable to the field f	fieldOfType
c methodsOfSig: s	matches methods with signature s that are implemented or overridden by the class c	methodsOfSig
c allMethodsOfSig: s	matches methods with signature s that are implemented, overridden or inherited by the class c	allMethodsOfSig
Behavioral Pattern Expr.	**Run-time Event Pattern(s) Matched**	**Abbreviation**
c callsTo: s receiver: r	matches method calls to methods with the signature s received by objects assignable to the type r in the control flow of instances of the class c,	callsTo
	`callsTo($c o): call($s) && target($r) && cflow(execs(o))`	
c callsReceived: s	matches method calls to methods with the signature s received by objects assignable to the class c	callsRec
	`callsRec($c o): call($s) && target(o)`	
mc valueOfArg: i	matches run-time values of the i^{th} argument of the method call mc	argVal
	`argVal(): $mc && args(.., $i, ..)`	
c argument: i ofCall: mc_1 sameAsArg: j ofCall: mc_2	matches if the i^{th} argument of the method call mc_1 points to the same object as the j^{th} argument of the method call mc_2, in the control flow of objects of the class c	argSameObj
	`argSameObj($c o): $argVal(mc2, j) && dflow[j, i] ($argVal(mc1, i)) && execs(o)`	
c methodCall: mc_1 before: mc_2	matches if in the control flow of instances of the class c, the method call mc_1 occurs at least once before the occurrence of method call mc_2	before
	`before($c o): execs(o) && ($mc1+ $mc2)`	
m returnedObjectTypes: c	matches all possible types of the objects returned by the method m from the point of view of the class c that implements, overrides, or inherits m	retTypes
	`retTypes(): execution($m) && returnTypes() && this($c)`	
f assignedNull	matches assignments to the field f with the `null` value	assignNull
	`assignNull(Object o): set($f) && args(o) && if(o == null)`	
f assignedNew: cs	matches assignments to the field f with an object returned by a constructor call with the signature cs	assignNew
	`assignNew(): set($f) && args(o) && dflow[o, i] (call($cs) && returns(i))`	
Helper Definitions	matches executions of methods in instances of class c	
	`execs($c o) : execution(* *(..)) && this(o);`	

Table 1: Types of structural and behavioral code patterns

the total number of code patterns that all instances of the feature f correspond to. The correspondence is specified using pattern expressions attached to the features. For a feature f let

– A_f be the number of all patterns in the code that satisfy the pattern expression,

– Q_f be the number of patterns matched by the query,

– C_f be the number of patterns that satisfy the pattern expression matched (correctly) by the code query,

– M_f be the number of patterns that satisfy the pattern expression missed by the code query (false negatives),

– I_f be the number of patterns that do not satisfy the pattern expression matched (incorrectly) by the code query (false positives).

The following two equations hold:

$$A_f = C_f + M_f \tag{1}$$

$$Q_f = C_f + I_f \tag{2}$$

Precision (P_f) and recall (R_f) can be defined as follows:

$$P_f = \frac{C_f}{Q_f} = \frac{C_f}{C_f + I_f} \tag{3}$$

$$R_f = \frac{C_f}{A_f} = \frac{C_f}{C_f + M_f} \tag{4}$$

For a feature f, value Q_f was returned by the prototypes at the end of reverse engineering for the code queries used by the prototypes. We then manually analyzed the code to determine the values for M_f and I_f for the given query. The analysis allowed us to propose the refined code queries that would capture the false negatives and exclude false positives of the previous query. Consequently, the values Q_f, M_f and I_f for the proposed code queries were obtained manually by checking wheather each false negative and false positive would belong to the results of the proposed query. Values C_f were calculated using equation 2. We present the details of the study in section 6.

5. CODE PATTERNS & CODE QUERIES

In order to retrieve framework-specific models, code patterns specified by the mapping of the abstract syntax to the framework API of an FSML must be matched in the framework completion code. Code patterns can be classified as structural or behavioral patterns. In general, structural patterns consist of code elements and their static properties as well as properties derived according to the static semantics, such as resolved type and method bindings. Because run-time events do not exist statically, behavioral patterns consist of *shadows* [13] of the run-time events over the code.

The types of code patterns identified in the mapping definitions of the three FSMLs are summarized in Table 1. The first column contains Smalltalk-like expressions that can be used to specify the patterns. The last column defines abbreviations used to refer to the given pattern type in the remainder of this paper. The second column presents descriptions of the semantics of code patterns. The description specifies the patterns in the code that match the given pattern expression. Since structural patterns can be fully re-

Pattern T.	Code query	Result	Abbreviation
callsTo	c getCallsInHierarchy: s receiver: r	a set of method calls with the signature s within the bodies of the class c and its superclasses, such that the receiver of each call is assignable to the type r	getCallsWH*
	c getCallsCFlow: s receiver: r	a set of method calls with the signature s in the control flow of every implemented, inherited, and overridden method of the class c, such that the receiver of each call is assignable to the type r	getCallsCF
callsRec	c getCallsReceived: s	a set of method calls with the signature s, such that the receiver of each call is assignable to the type c	getCallsRec*
	c getCallsReceivedTI: s	a set of method calls with the signature s, such that the receiver of each call is assignable to the type c. In the case when the type of the receiver is more general then the type c, the query traverses the receiver's dataflow graph backwards to infer its more specific type	getCallsRecTI
argVal	mc getArgValLiteralConstant: i	value of the i^{th} argument of the method call mc retrieved from a `static final` variable or a literal	getArgValLC*
	mc getArgValConstantProp: i	set of values of the i^{th} argument of the method call mc retrieved using interprocedural constant propagation limited in scope to the class that contains the called method	getArgValCP
	mc getArgValPartialEval: i	set of values of the i^{th} argument of the method call mc retrieved using partial evaluation	getArgValPE
argSameObj	c argument: i ofCall: mc_1 andArg: j ofCall: mc_2 isThis	true iff both the i^{th} argument of the method call mc_1 and the j^{th} argument of the method call mc_2 are the literal `this` and the resolved type of the literal is class c	argIsThis*
	c argument: i ofCall: mc_1 andArg: j ofCall: mc_2 isPrvFieldGivenCSeq: cs	true iff both the i^{th} argument of the method call mc_1 and the j^{th} argument of the method call mc_2 are references to the same private field of class c whose value has been assigned once before both calls	argIsPrvFieldAO
before	c is: mc_1 before: mc_2 inHierarchyGivenCSeq: cs	true iff the method calls mc_1 and mc_2 are located within the bodies of callback methods m_1 and m_2, respectively, such that the method m_1 occurs before the method m_2 in the callback sequence cs OR true iff mc_1 occurs before mc_2 in the cflow of the method m_1 if $m_1 = m_2$. Methods m_1 and m_2 can be any implemented, inherited or overridden methods of the class c	isBeforeWH*
	c is: mc_1 before: mc_2 inCFlowGivenCSeq: cs	true iff the method calls mc_1 and mc_2 occur in the control flows of callback methods m_1 and m_2, respectively, such that the method m_1 occurs before the method m_2 in the callback sequence cs OR true iff mc_1 occurs before mc_2 in the cflow of the method m_1 if $m_1 = m_2$. Methods m_1 and m_2 can be any implemented, inherited or overridden methods of the class c	isBeforeCF
retTypes	m returnStmsWithinAndSuper: c	a set of types of objects returned by the method m (excluding Object) retrieved from type bindings of return statements within the body of the method, including bodies of super methods if called. The type of the returned literal `this` is interpreted as class c	getRetTypesWS*
	m returnStmsMostSpecificType: c	a set of types of objects returned by the method m (excluding Object) retrieved from return statements, inferring the most specific type in the data flow of each returned object. The type of the returned literal `this` is interpreted as class c	getRetTypesMST
assignNew	f getAssignedNew: cc	a set of assignments to the field f with the constructor call cc	getAssgnNew*
assignNull	f getAssignedNull	a set of assignments to the field f with the `null` literal	getAssgnNull*

Table 2: Code queries for retrieval of behavioral patterns

trieved from the code by static analysis and their semantics are rather simple, we deem unnecessary a more formal definition in this paper. However, the semantics of behavioral patterns, which is more difficult to define, is specified more precisely using *meta pointcuts* in addition to the informal description.

Pointcuts were introduced in aspect-oriented programming [15] as expressions that define patterns of run-time events. In that context, crosscutting behavior can be applied when such patterns occur at run-time. In the context of FSMLs, pointcuts provide the exact definitions of the behavioral patterns that features correspond to. In Table 1, we use meta pointcuts parameterized with variables from the pattern expressions. The parameters of the meta pointcuts are prefixed with a `$` sign in order to distinguish them from other pointcut variables. We reuse elements of syntax of AspectJ [14] and some of its extensions, namely the Data Flow Pointcut [17] and Tracematches [1]. For example, the meta pointcut for the pattern type `callsTo` uses AspectJ's `call`, `target` and `cflow` pointcuts, and uses the `execs` helper pointcut (see bottom of Table 1); whereas the pattern type `argSameObj` uses `dflow` to specify that the argument of the first method call is the same object as the argument of the second call; and `before` uses the Tracematches notation to define the order in which method calls occur. However, current pointcut languages do not provide enough expressiveness and we needed to create a new primitive pointcut, namely `returnTypes`, used in the pointcut for the pattern type `retTypes`. This pointcut captures the run-time type of the object returned by a method.

The mapping definitions of the analyzed FSMLs use pattern expressions, whereas the prototype implementations of the FSMLs use code queries for matching the required code patterns. We present the code queries approximating behavioral patterns in Table 2. Queries marked with asterisk (*) are the ones used in the prototypes, while the remaining queries are the ones we propose as query refinements, which are one of the results of the second phase of the study. Code queries are defined in the Smalltalk-like notation, similar to their corresponding pattern expressions in Table 1. For each

FSML Feature	Pattern Expression	Query Type	Sun		Internet		Total		
			A_f	M_f	A_f	M_f	A_f	M_f	R_f
[0..*] Applet	class		20	0	51	0	71	0	100
![1] extendsApplet	assignableTo: Applet		20	0	51	0	71	0	100
[0..*] showsStatus	callsReceived: showStatus(String)	getCallsRec	11	0	28	0	39	0	100
[0..1] message	valueOfArg: 1	getArgValLC	5	4	18	2	23	6	73.91
		getArgValCP		3		2		5	78.26
		getArgValPE		0		0		0	100
[0..1] listensToMouse			10	0	13	0	23	0	100
![1] implementsMouseList[...]	assignableTo: MouseListener		10	0	13	0	23	0	100
![1] registers	callsReceived: addMouseListener([...])	getCallsRec	10	0	13	0	23	0	100
[1] deregisters	callsReceived: removeMouseListene([...])	getCallsRec	9	0	1	0	10	0	100
[1] deregistersSameObject	argument: 1 ofCall: ../../registers [...]	argIsThis	9	0	1	0	10	0	100
[1] registersBeforeDe[...]	methodCall: ../../../registers before: ../..	isBeforeWH	9	0	1	0	10	0	100
[0..*] thread	field		7	0	22	0	32	0	100
![1] typedThread	fieldOfType: Thread		7	0	22	0	32	0	100
[1] initializesThread	assignedNew: Thread(IRunnable)	getAssgnNew	7	0	20	0	30	0	100
[1] nullifiesThread	assignedNull	getAssgnNull	6	0	12	0	17	0	100
[0..*] parameter	callsReceived: getParameter(String)	getCallsRec	51	5	102	0	153	5	96.73
		getCallsRecTI		0		0		0	100
[0..1] name	valueOfArg: 1	getArgValLC	73	37	144	52	217	85	60.82
		getArgValCP		17		4		21	90.32
		getArgValPE		0		0		0	100

Table 3: Statistics for framework-specific models retrieved using Applet FSML

code query we provide a description of the results obtained by statically applying the query to the code.

Code queries presented in Table 2 can be grouped based on the kind of approximation they employ. We discuss potential false positives and false negatives for each group.

One group are queries that approximate interprocedural control flow graph of an object: `getCallsWH` and `isBeforeWH`. The idea is to search in the body of the class of the object and its superclasses because the code implementing the object is likely to be found there. Those queries can potentially miss patterns (false negatives) located in helper classes whose code is located outside the class of the object (local and anonymous classes are included in the search). Also, the queries can incorrectly identify patterns (false positives) in the superclasses, which reside in the bodies of overridden methods, which are not called using `super`.

Another group of code queries are queries that rely on the information about method callback sequence of the framework: `isBeforeWH/CF` and `argIsPrvField`. The callback sequence information is necessary because the control flow graph of a class implementing callback methods is potentially composed of disjoint graphs for each callback method, unless the callback methods call each other, which is not common. The query `isBeforeWH` will miss a pattern if at least one of the method calls is in the control flow of a callback method, but not directly in the body of that method. Similarly, the query `argIsPrvField` has to determine whether the only field assignment occurred before the first method call. The query is motivated by a very common programming pattern, whereby an instance of a helper class is created and assigned to a private field and then used as a parameter of service method calls. These queries will lead to false negatives in cases where patterns cannot be traced back to field initializations, a constructor or a callback method, for which the precedence is known.

Another group are queries that traverse the dataflow graph backwards beginning at a particular use of a variable. The query `getCallsRecTI` determines the most specific type of the method call receiver and therefore it can potentially match patterns missed by the query `getCallsRec` which only uses the static type binding of the receiver. Analogously, the query `getRetTypesMST` determines the most specific type of the returned variable and therefore it can potentially match patterns missed by the query `getRetTypesWS` which only uses the static type binding of the return expression. The query `getRetTypesWS` will return an inappropriate type if the object to be returned i) is assigned to a variable with more general type than the object's type and the variable is returned or ii) the object is returned by a method called from the return statement with more general return type than the object's type. The query `getArgValCP` locates all static values that a variable in question can assume by traversing the dataflow graph of that variable.

The queries `getAssgnNew` and `getAssgnNull` only match patterns in which the right-hand side of a field assignment is the `new` expression or the `null` literal. These queries will miss patterns where an intermediate variable is assigned first and then the field is assigned with the variable. In these cases, dataflow graph traversal is also necessary.

6. PRECISION & RECALL DATA

Tables 3, 4, and 5 present values A_f, M_f, and R_f for every code query used for the retrieval of the feature f. Each feature in the column *FSML Feature* is associated with a pattern expression presented in the column *Pattern Expression*. We provided values for some parameters of pattern expressions to give the reader an idea about the meaning of the features. We used [...] to indicate omitted details. The complete mapping definitions can be found elsewhere [3]. Features that do not have a pattern expression are abstract and are used solely for the purpose of grouping other features (e.g., `listensToMouse` in Table 3). Pattern expressions `class` and `field` indicate that a feature corresponds to a Java class or a field, respectively. The properties of classes or fields a feature will correspond to are specified by the feature's essential subfeatures. Values of the parameters

FSML Feature	Pattern Expression	Query Type	Roller		Mailreader		Cookbook		Total		
			A_f	M_f	A_f	M_f	A_f	M_f	A_f	M_f	R_f
[0..*] Action	class		58	0	19	0	16	0	93	0	100
![1] extendsAction	assignableTo: Action		58	0	19	0	16	0	93	0	100
[0..1]extendsDispatch[...]	assignableTo: DispatchAction		36	0	11	0	0	0	47	0	100
[0..*] actionMethod	methodsOfSig: *(Action[...], [...], [...], [...])		114	0	10	0	0	0	124	0	100
[0..1]overridesExecute	methodsOfSig: execute([...], [...], [...], [...])		14	0	13	0	15	0	42	0	100
[0..*] forwardImpl	callsTo: findForward(String)	getCallsWH	186	0	5	0	21	0	212	0	100
[1] name	valueOfArg: 1	getArgValLC	186	0	4	0	21	0	211	0	100

Table 4: Statistics for framework-specific models retrieved using Struts FSML

of the pattern expressions can be either statically provided in the metamodel or can be retrieved from parent features during reverse engineering. For example, the pattern expression `assignableTo: Applet` for the feature `extendsApplet` requires a Java class as the value of the parameter `c`, which, in this case, will be the class that the parent feature `Applet` will correspond to. Values of the parameters can also be the patterns that other features correspond to, in which case, the features need to be specified using path expressions. For example, the pattern expression attached to the feature `deregistersSameObject` requires two method calls and uses paths "`../../registers`" and "`..`" to retrieve calls that the features `registers` and `deregisters` correspond to.

The column *Query Type* contains names of code queries used for retrieving patterns of the given type. We did not include queries for structural patterns for brevity. For behavioral patterns, we provide queries that were used by the prototypes. If a query retrieved less than 100% of patterns, we included data for queries we proposed in the subsequent rows. The last column, R_f, contains the recall calculated according to the equation 4. Except for one case, the precision is always 100% and we did not include it in the tables. We discuss the data in Section 7.1.

7. DISCUSSION

The target use scenario for automatic extraction of framework-specific models is to help application developers understand how a framework is used by their application. Framework-specific models are therefore non-essential during the development and are not considered as a primary development artifacts. They can, however, provide benefits to both the developers and the designers [3]. In this context, using FSMLs can be characterized as *model-supported engineering* rather than *model-driven engineering*.

7.1 Precision & Recall

We discuss the data presented in Tables 3-5, each table separately. We focus on the highlighted cells.

Surprisingly, for all except one features and the code queries used in the prototypes the precision turned out to be 100%. Precision depends on false positives and by checking all features from the retrieved models we concluded that only one was a false positive. As we discussed in Section 5 all of the code queries can potentially return false positives. Therefore, finding only one false positive in the models for the analyzed applications only means that the particular applications we have chosen were written in a way that the queries did not return many false positives.

Table 3. *The feature* `message`. The 6 values missed by the first query were neither string literals nor static final variables. One more value could be retrieved by constant propagation and all remaining values could be retrieved by partial evaluation (string concatenation). 16 values were not retrieved because the value cannot be determined statically. We did not count these values as false negatives.

The features `deregistersSameObject` and `registersBeforeDeregisters`. In all cases, both the registration and deregistration calls used the literal `this` as an argument, and all registration and deregistration calls were located in the `init` and `destroy` methods, respectively. Both methods are callback methods and `init` is called before `destroy`.

The feature `thread`. 32 fields of type `Thread` were found. The reason why only 30 fields are initialized is that two applets declared two fields which were never used. Also, we did not find any false negatives for the queries `getAssgnNew/Null`, meaning that in all cases the right hand side of a field assignment was a constructor call or the literal `null`.

The feature `parameter`. The 5 missed calls were located in the constructor of a helper class and the constructor's parameter `applet` was the receiver of the calls. The helper class is instantiated twice by the applet and the literal `this` is used as a parameter to the constructor. Therefore, query `getCallsRecTI` would infer that the applet is, in fact, the receiver of the 5 method calls.

The feature `name`. The 85 missed parameter names can be retrieved using constant propagation and loop unrolling. In 3 cases (for 3 instances of feature `parameter`), a call to `getParameter` was placed in a helper method, which was then called 64 times with static values. Traversal of the dataflow graph with the distance of at most 2 method calls was necessary to reach the static values. In 2 cases (for 2 features), a call to `getParameter` was placed in a loop with a statically known loop count. In the first case, the static values of the method call parameter were constructed by appending the loop count variable to a constant string and loop unrolling would yield 4 values. In the second case, the static values were retrieved from a static array using the loop count variable as index. Again, loop unrolling would yield additional 17 values. For 11 features, the static value cannot be determined and these are not false negatives.

Table 4. *The feature* `name`. In the three example applications, the developers used either string literals or `public static final` fields as arguments of the method call. The reason is that the names used as parameters of the `findForward` method calls must match the names of forward declarations in Struts' XML configuration file. The single value that was not retrieved comes from a HTTP request and we did not count it as a false negative.

Table 5. *The concept* `SelectionListener`. The 8 workbench parts are selection listeners. In particular, one is a global selection listener, six are global post selection listeners, and one is a specific selection listener.

FSML Feature	Pattern Expression	Query Type	org.eclipse.pde.ui		
			A_f	M_f	R_f
[0..*] Part	class		88	0	100
![1] implementsIView/IEditorPart	assignableTo: IViewPart/IEditorPart concreteOnly: true		88	0	100
[0..*] SelectionProvider	class		1	0	100
![1] implementsISelectionProvider	assignableTo: ISelectionProvider		1	0	100
[1]registers	callsTo: setSelectionProvider(ISelectionProvider)	getCallsWH	1	0	100
[0..*] SelectionListener	class		8	0	100
![1] implementsISelectionListener	assignableTo: ISelectionListener		8	0	100
[0..1] globalSelectionListener	callsTo: addSelectionListener(ISelectionListener)	getCallsWH	1	0	100
[1] deregisters	callsTo: removeSelectionListener(ISelectionListener)	getCallsWH	1	0	100
[1] deregistersSameObject	argument: 1 ofCall: ../.. sameAsArg: 1 ofCall: ..	argIsThis	1	0	100
[1] registersBeforeDeregisters	methodCall: ../../.. before: ../..	isBeforeWH	1	0	100
[0..1] globalPostSelectionListener	callsTo: addPostSelectionListener(ISelectionListener)	getCallsWH	6	0	100
[1] deregisters	callsTo: removePostSelectionListener(ISelectionListener)	getCallsWH	6	0	100
[1] deregistersSameObject	argument: 1 ofCall: ../.. sameAsArg: 1 ofCall: ..	argIsThis	6	0	100
[1] registersBeforeDeregisters	methodCall: ../../.. before: ../..	isBeforeWH	4	0	100
[0..*] specificSelectionListener	callsTo: addSelectionListener(String, ISelectionListener)	getCallsWH	1	0	100
![1] registrationPartId	valueOfArg: 1	getArgValLC	1	0	100
[1] deregisters	callsTo: removeSelectionListener(String, ISelectionLi[...])	getCallsWH	1	0	100
[1] deregistrationPartId	valueOfArg: 1	getArgValLC	1	0	100
[1] deregistersSameObject	argument: 2 ofCall: ../.. sameAsArg: 2 ofCall: ..	argIsThis	1	0	100
[1] registersBeforeDeregisters	methodCall: ../../.. before: ../..	isBeforeWH	0	0	100
[0..*] PartListener	class		10	0	100
![1] implementsIPartListener	assignableTo: IPartListener		10	0	100
[1] registers	callsTo: addPartListener(IPartListener)	getCallsWH	10	0	100
[1] deregisters	callsTo: removePartListener(IPartListener)	getCallsWH	10	0	100
[1] deregistersSameObject	argument: 1 ofCall: ../../reg[...] sameAsArg: 1 ofCall: ..	argIsThis	16	6	62.5
		argIsPrvFieldAO		0	100
[1] registersBeforeDeregisters	methodCall: ../../../registers before: ../..	isBeforeWH	10	6	62.5
		isBeforeCF		0	100
[0..*] AdapterProvider	class		44	0	100
![1] providesAdapter	allMethodsOfSig: Object getAdapter(Class)		44	0	100
![1..*] adapters	returnedObjectTypes	getRetTypesWS	191	59	69.11
		getRetTypesMST		0	100
[0..*] AdapterRequestor	class		22	0	100
![1..*] requestsAdapter	callsTo: getAdapter(Class) receiver: IWorkbenchPart	getCallsWH	69	0	100
[1] adapter	valueOfArg: 1	getArgValLC	62	0	100

Table 5: Statistics for a framework-specific model retrieved using WPI FSML

The features `deregistersSameObject`. All patterns were matched because the literal `this` was used in both the registration and deregistration calls.

The features `registersBeforeDeregisters` (of selection listeners). The 3 patterns not matched by the query (2 for post selection listeners and 1 for specific selection listener) are not false negatives because the order of method calls cannot be determined statically: the registration and the deregistration calls are invoked from the UI actions.

The concept `PartListener`. *The features* `deregistersSameObject` and `registersBeforeDeregisters`. All part listeners inherit behavior from an abstract view, where the literal `this` is used in the registration and the deregistration. Both calls occur in the `createPartControl` and `dispose` methods, which are callback methods. Both calls are not false positives because all part listeners delegate to `super` in `createPartControl` and `dispose` methods, which they override. Six of the part listeners inherit additional registration and deregistration calls from another abstract view, which registers and deregisters an instance of a helper part listener. The instance of the helper listener is stored in a private field and is assigned only once before the registration. The registration occurs in the cflow of `createPartControl` and the deregistration occurs in the cflow of `dispose` but not in their bodies, which is why the pattern was missed by `beforeWH`.

The concept `AdapterProvider`. *The feature* `adapters`. The 59 patterns missed by the query `getRetTypesWS` can be divided into two categories, whereby i) the return statement delegates to a factory method and ii) the return statement returns a variable. In the first category, the factory method's return type is more general then the type of the returned object. In the second category, the type of the variable is more general than the type of the returned object assigned to the variable. The query `getRetTypesMST` captures all patterns by analyzing the dataflow of the returned object, beginning at the return statement, and inferring the most specific type of the object. In ten cases the exact type could not be found because of polymorphic calls (in most of these cases, the type of the receiver was an interface).

The concept `AdapterRequestor`. *The features* `requestsAdapter` and `adapterType`. The 7 adapter requestors inherit the adapter request call from an abstract multi-page editor class, where the editor simply delegates the call to a page with an active editor. The argument value cannot be statically determined and we did not count these cases as false negatives. The only one false positive is because an editor overrides a method from the abstract superclass that contains the adapter request call and does not call super.

It is important to note that even if a value of an argument of a method call cannot be statically determined, a framework-specific model still provides traceability to the method call. In the case of the Struts FSML, where retrieving the names of forwards is critical for referential integrity checking with the XML configuration file, the results show that such values are statically available in the code.

Finally, the weighted average recall for queries that missed patterns is 82.33% for `getArgValueLC`, 82.35% for `argIsThis`, 76% for `isBeforeWH`, and 97.77% for `getCallsRec`, which shows that even such simple queries provide very high recall. An exception here is the query `getRetTypesWS` with recall 69.11%. However, we counted a false negative if the query returned a more general type than the actual type of the returned object, which, in all cases, was an interface. Even returning a more general type provides far more information than the return type of the method (`Object` in the case of `getAdapter()`) and, in fact, is sufficient for WPI FSML because workbench parts usually request an adapter implementing a certain interface.

7.2 Threats to validity

We discuss the limitations of our study in terms of threats to the validity of the obtained results and describe measures undertaken in order to minimize such threats. We distinguish between *internal validity*, in which the elements that might compromise the design and analysis of the study are discussed, from *external validity*, which relates to the extent to which conclusions can be generalized [16].

Internal Validity. The main threat to internal validity is related to the measurement procedures. The queries implemented in the prototypes matched patterns in the code. There are two situations in which errors in the queries' implementations may influence the results: i) a false negative pattern is matched by the query and ii) a false positive pattern is missed by the query. In the first case, the recall appears higher than in reality and in the second case the precision appears higher than in reality. A similar problem can emerge in the determination of the Q_f and A_f values for the newly proposed queries, which was performed by manual code inspection. If patterns were missed, the results would indicate precision and recall values greater than they really are. Both threats were minimized by having two of the authors independently collect and compare the data, and by supporting the inspection of code and protype results with two code query tools: JQuery [9] and the built-in Eclipse Java Development Tools [10] search.

External Validity. Our study involves three input variables: frameworks, FSMLs, and applications. The way in which instances were selected for these variables directly affects the external validity of our results.

Frameworks and FSMLs. The construction of an FSML involves selecting and modeling some concepts in the area of interest. Consequently, the results are restricted not by the frameworks and FSMLs themselves, but by the characteristics of the chosen concepts. For example, highly dynamic concepts of frameworks such as Java Swing prescribe the construction of complex object structures, which are difficult to analyze statically.

Applications. The selection of representative applications for each framework directly influences the results of our study because the precision and recall values are highly dependent on how the applications use the framework. In order to obtain results that can be generalized, we chose not only applications that were provided by the framework developers, but also other applications at random. For example, we used example Applets and Struts applications provided with the frameworks, and core eclipse plug-ins, but added other applications acquired at random from the Internet. The goal is to guarantee that our results are not biased by using only applications that strictly follow the framework examples. Furthermore, our sample consists of applications that directly use the framework. We consider the construction of custom layers on top of a base framework equivalent to the construction a new framework and therefore new definitions of FSML concepts are necessary.

7.3 Empirical approach to query refinement

Our results suggest an empirical approach to code query refinement, whereby the categorization of false negatives and false positives of a given query allows extending the query such that the false positives from a given category are always missed and false negatives are retrieved. A good example from our study is the query `isArgPrvFieldAO`. This is in contrast to the general approach, in which the pursuit of *soundness* and *completeness* requires using very expensive analyses. The guidelines for developers (e.g., the *monkey see, monkey do* rule) and recent research on design fragments [11] suggest that the developers commonly copy existing examples and utilize common programming micropatterns when using frameworks. Therefore, empirical query refinement could result in efficient code queries that provide high precision and recall when applied to real code.

8. RELATED AND FUTURE WORK

In this section we describe related works grouped in the following categories.

General Design & Architecture Recovery. The main difference between the general design and architecture recovery tools and a framework-specific approach is that the latter heavily relies on the framework knowledge, which on the one hand allows the retrieval of meaningful and precise models, but on the other hand requires designing an FSML for each framework. A detailed comparison between framework-specific and general-purpose design retrieval remains future work.

Generic code query tools. Generic code query tools for Java, such as JQuery [9], JTL [7], and CodeQuest [12] cannot query for the kinds of behavioral patterns required for the retrieval of framework-specific models. In particular, the dynamic pattern types presented in Table 1 cannot be retrieved. Another difference is that generic code query tools usually build a complete database of facts about the queried program, which, as shown in Section 2 is not necessary. The only types of patterns that such tools could provide without incurring a prohibitive increase in the size of the fact database are patterns matched by the queries `getArgValLC`, `getAssgnNull`, and `getAssgnNew`.

Static analysis frameworks. Static analyzers usually build a complete control flow graph of the application, which is a prerequisite for many other static analyses. As discussed in Section 2, the analysis must be performed on-demand and in the presence of incomplete programs. The following two works deal with the static analysis of framework-based code. Component Level Dataflow Analysis [19] is an approach to analysis of a program in the presence of large

libraries, where only the program is analyzed and the analysis relies on the availability of summary information about the library/framework. Zhang et al. propose an algorithm for computing a call graph of an application in the presence of callback methods [21].

Defining framework-provided concepts. We are not aware of any systematic approach to defining framework-provided concepts for the purpose of reverse engineering other than the FSML approach [3]. Also, we are not aware of any work proposing the specification of the correspondence between model elements and code patterns using pointcuts.

Aspect Weaving Optimization. An active research topic in the aspect-oriented programming community is the optimization of the run-time performance of aspect-oriented programs by removing unnecessary run-time checks, e.g., [6]. Such optimization techniques perform static analysis to determine whether certain shadows will always or never be executed when the given pointcut matches. Unfortunately, such analyses tend to require the complete control flow graph of the application and thus are not applicable in the context of FSMLs for the reasons discussed in Section 2. Therefore, while advances in weaving optimization could be leveraged by FSMLs, currently the only feasible solution is to use approximations in the form of code queries. We do, however, believe, that the techniques used in dynamic pointcut weaving optimization can be leveraged to design code queries that provide the highest possible precision and recall.

9. CONCLUSION

Framework-specific models describe how framework-provided concepts are implemented in the application code. Automatic location of concept instances requires matching structural and behavioral patterns in the code, which can be realized by code queries. In this paper we evaluated the precision and recall of code queries that can be used for model retrieval.

We identified the types of structural and behavioral patterns that features of the three example FSMLs correspond to and provided a more precise definition of behavioral patterns using meta pointcuts. We showed how a priori knowledge about a framework can be leveraged for the retrieval of behavioral patterns, such as `callsTo`, `before`, `argSameObj`. Also, we provided empirical evidence that by leveraging framework knowledge and concentrating on the framework boundary simple static analyses are sufficient for retrieving framework-specific models, without requiring whole-application analysis. The average recall for all simple queries for behavioral patterns is 88.06% and the precision is 99.93%. We proposed refined versions of the code queries that would achieve 100% precision and recall when analyzing the example applications. Finally, the discussed queries suggest improvements to general-purpose code query tools to make them more usable for the retrieval of framework-specific models.

Acknowledgments. This work is partially supported by IBM Centers for Advanced Studies, Ottawa and Toronto. We thank George Fairbanks for providing example applets.

10. REFERENCES

[1] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. *SIGPLAN Not.*, 40(10):345–364, 2005.

[2] M. Antkiewicz and K. Czarnecki. Framework-specific modeling languages with round-trip engineering. In *MoDELS*, volume 4199 of *LNCS*, pages 692–706, 2006.

[3] M. Antkiewicz and K. Czarnecki. Framework-specific modeling languages; examples and algorithms. Technical Report 2007-18, ECE, U. of Waterloo, 2007.

[4] Apache Software Foundation. *Roller Weblogger 3.0*. `http://rollerweblogger.org/`.

[5] Apache Software Foundation. *Struts User's Guide*. `http://struts.apache.org/1.3.8/index.html`.

[6] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Optimising AspectJ. In *PLDI'05*, pages 117–128, 2005.

[7] T. Cohen, J. Y. Gil, and I. Maman. JTL: the Java tools language. In *OOPSLA'06*, pages 89–108, 2006.

[8] K. Czarnecki and U. W. Eisenecker. *Generative programming: methods, tools, and applications; Appendix A: Conceptual Modeling*, pages 721–737. ACM Press/Addison-Wesley Publishing Co., 2000.

[9] K. De Volder. JQuery: A generic code browser with a declarative configuration language. In *PADL'06*, volume 3819 of *LNCS*, pages 88–102, 2006.

[10] Eclipse Foundation. *Java Development Tools*. `http://www.eclipse.org/jdt/`.

[11] G. Fairbanks, D. Garlan, and W. Scherlis. Design fragments make using frameworks easier. In *OOPSLA'06*, pages 75–88, 2006.

[12] E. Hajiyev, M. Verbaere, and O. de Moor. CodeQuest: Scalable source code queries with datalog. In *ECOOP'06*, volume 4067 of *LNCS*, pages 2–27, 2006.

[13] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In *AOSD '04*, pages 26–35, 2004.

[14] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP'01*, pages 327–355, 2001.

[15] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97*, pages 220–242, 1997.

[16] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. E. Emam, and J. Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Trans. Softw. Eng.*, 28(8):721–734, 2002.

[17] H. Masuhara and K. Kawauchi. Dataflow pointcut in aspect-oriented programming. In *APLAS'03*, volume 2895 of *LNCS*, pages 105–121, 2003.

[18] C. Pandit. *Make your Eclipse applications richer with view linking*, 2005. `http://www-128.ibm.com/developerworks/opensource/library/os-ecllink/`.

[19] A. Rountev, S. Kagan, and T. J. Marlowe. Interprocedural dataflow analysis in the presence of large libraries. In *CC'06*, volume 3923 of *LNCS*, pages 2–16, 2006.

[20] Sun Microsystems. *Java Tutorials, Lesson: Applets*. `http://java.sun.com/docs/books/tutorial/deployment/applet/index.html`.

[21] W. Zhang and B. G. Ryder. Constructing accurate application call graphs for Java to model library callbacks. In *SCAM 2006*, pages 63–74, 2006.