Autom. Soft. Eng. manuscript No.
(will be inserted by the editor)

Fast Extraction of High-Quality Framework-Specific Models from Application Code

Michał Antkiewicz · Thiago Tonelli Bartolomei · Krzysztof Czarnecki

Received: date / Accepted: date

Abstract *Framework-specific models* represent the design of application code from the framework viewpoint by showing how framework-provided concepts are instantiated in the code. Retrieving such models quickly and precisely is necessary for practical model-supported software engineering, in which developers use design models for development tasks such as code understanding, verifying framework usage rules, and round-trip engineering. Also, comparing models extracted at different times of the software lifecycle supports software evolution tasks.

We describe an experimental study of the static analyses necessary to automatically retrieve framework-specific 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 an open-source framework. For reverse engineering, we use prototype implementations of the three FSMLs.

Our results show that for the considered frameworks and a large body of application code rather simple code analyses are sufficient for automatically retrieving framework-specific models with high precision and recall. Based on the initial results, we refine the static analyses and repeat the study on a larger set of applications to provide more evidence and confirm the results.

This paper is an extended version of the paper "Automatic extraction of framework-specific models from framework-based application code", which was published in the proceedings of the Twenty-Second ACM/IEEE International Conference on Automated Software Engineering, 2007.

M. Antkiewicz, T. Tonelli Bartolomei, K. Czarnecki
Generative Software Development Lab
University of Waterloo
200 University Ave. West
Waterloo, N2L 3G1, ON, Canada
E-mail: {mantkiew, ttonelli, k2czarne}@uwaterloo.ca
`http://gsd.uwaterloo.ca`

The refined static analyses provide precision and recall of close to 100% for the analyzed applications.

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 that can be completed to build custom applications. An object-oriented framework consists of core classes that implement core functionality and application programming interface (API) classes that can be extended or instantiated in the application code. Domain concepts provided by a framework are implemented in the application code by writing *framework completion code* according to the framework's API. Framework completion code is the code that implements the difference in functionality between the framework and the application, that is, it completes the framework. Unfortunately, the concept instances are often not easily recognizable by developers directly in the completion code because they are scattered across the code, tangled with each other and with instances of other concepts, and buried in a large amount of implementation detail. *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 [4–6]. Such models explicitly represent the instances of framework-provided concepts that are implemented in the completion code.

One way of formalizing framework-provided concepts is by decomposing them into hierarchies of *features* [4]. Features are distinguishing characteristics (properties) of a concept and they allow discriminating among concept instances. Features may correspond to *structural* or *behavioural patterns* in the application code. Matching 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 and recall by simple code queries, precisely determining matches of behavioural patterns in the application code using static analysis is potentially undecidable.

We report on a study that we conducted to measure the precision and recall of reverse engineering using code queries that locate instances of behavioural patterns in the completion code. The study was executed in three phases:

1. identification of types of patterns and their corresponding code queries;
2. evaluation of the precision and recall of the code queries and proposing a set of refined code queries; and
3. evaluation of the precision and recall of the refined code queries.

In the first phase, we analyzed code pattern definitions attached to features in three sample *framework-specific modeling languages* (FSMLs) [4]. FSMLs are domain-specific modeling languages that are designed for a specific area of concern of an object-oriented framework and are used for expressing

framework-specific models. The FSMLs used in this study were designed for: (i) Java Applet framework [31], (ii) Apache Struts framework [9], and (iii) a part of the Eclipse Workbench framework [27]. The result of the analysis is a classification of (i) patterns that the features correspond to and (ii) code queries that were implemented in the prototypes of the FSMLs in order to detect these patterns in application code.

In the second phase, we used the prototype implementations of the three FSMLs to reverse engineer a large number of sample applications built on top of the three frameworks. We then manually verified the correctness of the retrieved framework-specific models and calculated precision and recall of the used code queries. Additionally, we categorized common false positives and false negatives of the code queries and proposed a refined set of queries that would reach 100% precision and recall for the studied applications [5].

In the third phase, we implemented the refined code queries, slightly extended the three FSMLs, and repeated the study for an even larger number of applications in order to gather more evidence and confirm the results.

One 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 primary development artifacts. They can, however, provide benefits to both the developers and the designers [4]. In this context, using FSMLs can be characterized as *model-supported engineering* rather than *model-driven engineering*.

The intended audience of this paper is primarily tool builders interested in retrieving framework-specific models from application code. They can learn about specific approximations of behavioural patterns and their effectiveness. Also, the retrieval of models from framework completion code is of interest to the reverse engineering community. 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, FSML designers can learn about different types of structural and behavioural code patterns that can be used to define the mapping between models and code.

The main contribution of this paper is providing evidence that fast retrieval of high-quality models that represent the dynamic interaction between application code and frameworks is feasible using static analysis. We argue that by concentrating on the static framework boundary, which consists of all places in the application that interact with the framework, and by leveraging framework-specific knowledge (e.g., order of callbacks), rather simple code queries become sufficient. We provide evidence that the refined code queries provide precision and recall of close to 100% and do not incur a prohibitive increase in the analysis time. Furthermore, we give precise definitions of behavioural code patterns using meta pointcuts. We provide code queries that are approximations of the behavioural patterns and can be used for retrieving them. Finally, we discuss possible false positives and false 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 the static analysis of framework completion code. In Section 3, we provide the necessary background information on framework-specific models and FSMLs. Next, in Section 4, we describe the setup of the study, which is followed by Sections 5 and 6, where we provide the resulting data. In Section 8, we discuss the data and the threats to validity. We present related work in Section 9 and conclude the paper in Section 10.

2 Challenges of statically analyzing completion code

Framework-specific models describe how concepts provided by the framework are instantiated in the completion code. Concept instances are characterized by configurations of features and the features correspond to structural and behavioural patterns in the completion code. Therefore, automatic extraction of framework-specific models requires matching the structural and behavioural patterns in 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 by calling *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 the basis for many static analyses, is often infeasible.

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 *static framework boundary*, that is, all places in the code where the application interacts with the framework. The static framework boundary consists of all callback methods implemented in the application and all references to the framework code from the application. For example, all method calls to framework methods and all usages of framework types belong to the boundary.

Another characteristic of framework-based code is the use of configuration files, which are declarative specifications interpreted by a framework and which also belong to the framework boundary. 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. For example, any Java class can be assigned the role of a *bean* in the Spring framework [2] or a method can be assigned the role of an *action method* in Java Server Faces [30]

framework. Sometimes understanding the final behaviour of the application requires analyzing both the code and the configuration file in conjunction. For example, in Enterprise Java Beans 3 [17] the name of a bean, which is by default the name of the class, can be overridden using an appropriate Java annotation or using a configuration file. Hence, determining the final name requires interpreting the code and the configuration file with respect to the name override rules.

Therefore, the retrieval of framework-specific models requires both using configuration files and 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 or data flow graphs.

Given these challenges of 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 behavioural patterns, that is, they can miss some pattern instances in the code and they can match some parts of the code incorrectly. However, by allowing misses and incorrect matches, we are able to use simple analyses that scale to large bodies of code. At the same time, our study shows that only very few actual misses and incorrect matches occur for a large set of the analyzed applications.

In the next section, we describe how framework-specific models correspond to code patterns and how the correspondence can be defined using FSMLs.

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 sample 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 differ in terms of the implementations steps they entail. For example, in order to implement an instance of the concept *action*, the Struts framework prescribes subclassing an appropriate framework class and creating appropriate declarations in an XML configuration file. Furthermore, 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 different sets of attributes need to be set in XML action

declarations for different kind of actions. Therefore the instances of the concept *action* are not uniform: each instance can be characterized by a different set of *features*.

Framework-provided concepts can be captured as framework-specific modeling languages (FSMLs) [4]. The abstract syntax of an FSML is defined as a *feature model*, which decomposes 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 where some features are present. In the feature hierarchy, features can be *essential*, *mandatory*, or *optional* with respect to their parent feature. In a feature configuration, a parent feature cannot exist without all of its essential subfeatures (essential features represent the essential properties of the parent feature without which the parent feature cannot exist). Mandatory subfeatures should be present for their parent feature (it is a configuration error if a mandatory feature is missing), and optional subfeatures may or may not be present. Finally, a feature may also have a *type*, meaning that a value of that type can be associated with the feature in the configuration.

We say that the completion code is correct with respect to the framework's API if it satisfies API constraints. Since the abstract syntax of an FSML formalizes API constrains, we can check the conformance of the code to the API by checking the conformance of the extracted framework-specific model to the FSML's abstract syntax. The distinction between *essential* and *mandatory* is critical for being able to analyze incorrect code and represent concept instances in which mandatory parts of the implementation are missing or certain constraints are violated. Patterns of features of incorrect concept instances that are present in the code are expected to be matched by the FSML definition; however, matching of a candidate concept instance is stopped as soon as an essential feature is found missing. Essential features, therefore, are the minimum characteristics required to match a concept instance, even if the instance is incorrect.

Table 1 shows a fragment of the decomposition of the concept *action* into a hierarchy of features and sample feature configurations of two action instances. The hierarchy of features is represented using indentation (subfeatures are further right). Cardinalities of features are indicated in square brackets and are interpreted as follows: [0..1] for optional features, [1] for mandatory and essential features, and [0..*] and [1..*] for multiple features (i.e., those of which multiple copies can be included in a configuration [12]). Essential features are marked using the exclamation mark (!). The abstract syntax tree of a framework-specific model is a feature configuration which conforms to the feature hierarchy. Such a model contains instances of selected features, possibly many instances of multiple features, and values associated with features with types. For example, in the second and third column of Table 1 we can see two instances of the concept `Action`, respectively. The feature `name` has values `auth.Login` and `auth.Logout` in each configuration, respectively. The feature `extendAction` is present in both configurations because it is an essential

Concept Features	Concept Instance Feature Configurations	
[0..*] Action	Action	Action
[1] name (String)	name('auth.Login')	name('auth.Logout')
![1] extendsAction	extendsAction	extendsAction
[0..1] extendsDispatchAction	extendsDispatchAction	
[0..*] actionMethod (String)	actionMethod('main')	
	actionMethod('home')	
[0..1] overridesExecute		overridesExecute
[0..*] forward	forward	forward
[1] name (String)	name('success')	name('homePage')
	forward	
	name('accessDenied')	

Table 1 The concept *action* and some of its features (first column) and two sample action instances represented by feature configurations (second and third column).

feature, but the feature `extendsDispatchAction` is only present in the configuration for the Login action. When reading the feature configuration of the Login action, we immediately see that (i) the action is a *dispatch action*, (ii) it has two *action methods* called `main` and `home`, and (iii) it uses two forwards: `success` and `access denied`. In contrast, the Logout action is a regular action, it overrides the `execute` method (which is the default action method), and it uses one forward: `homePage`. If the mandatory subfeature `name` of the feature `forward` was missing, it would indicate an error in feature configuration but the feature `forward` would still be present. However, if the essential feature `extendsAction` was missing, it would have excluded its parent `Action` from the configuration because a parent feature (or concept) cannot exist without any of its essential subfeatures. More information on feature modeling and their interpretation in logics can be found elsewhere [13].

Features describing a concept instance can correspond to structural and behavioural patterns in the completion code that implements the instance. Figure 1 presents sample code that can be described by the first configuration from Table 1. Underlined code patterns correspond to feature instances and their values; we did not underline the entire class for readability. For example, the first configuration of `Action` from Table 1 corresponds to the Java class (lines 3-14). The feature `dispatchAction` corresponds to the superclass declaration (line 3). Note that the feature `extendsAction` is present because the class `Action` is a superclass of the class `DispatchAction`. The features `actionMethod` correspond to the method declarations (lines 4-11 and 12-13) and their values correspond to the names of the methods. The features `forward` correspond to method calls used to find forwards returned by action methods (lines 8 and 10) and the features `name` correspond to the values of method call arguments (lines 6 and 10). 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* (not shown).

```
1  package auth;
2  ...
3  public class Login extends DispatchAction {
4    public ActionForward main(ActionMapping mapping, ActionForm form,
5      HttpServletRequest request, HttpServletResponse response) {
6      String denied = "accessDenied";
7      if (...)
8        return mapping.getForward(denied);
9      ...
10     return mapping.getForward("success");
11   }
12   public ActionForward home(ActionMapping mapping, ActionForm form,
13     HttpServletRequest request, HttpServletResponse response) { ... }
14 }
```

Fig. 1 Sample code described by the first configuration from Table 1

The *mapping* of a feature model defining the FSML abstract syntax to the framework API defines the correspondence between the features and the code patterns [4]. Each feature in the feature model has a *mapping definition* attached, which exactly specifies the code patterns that can correspond to the feature instance in a feature configuration. Mapping definitions use predefined *mapping types* that represent basic kinds of feature-to-code-pattern correspondences: mapping definitions specify values of the parameters defined by mapping types. For example, a mapping type defines a correspondence to methods of a class and a mapping definition specifies exactly the signature of the methods to be matched. Table 4 in Section 5 presents mapping definitions for the concept `Action` and its features. In the remainder of the paper we refer to a feature model of abstract syntax together with mapping definitions as the *metamodel* of an FSML.

The mapping definitions are only declarative specifications of the correspondence and they cannot be directly executed. In our implementation, we realized mapping types by implementing *code queries* and *code transformations* for each type. 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. The execution of code queries and transformations is controlled by parameters specified in mapping definitions. For example, by looking at the essential feature of the concept *action*, we immediately see that instances of *action* can be identified by a code query returning classes assignable to the Java class *Action*. A code transformation can create or remove Java classes implementing instances of the concept *action*. The entire 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 [4, 7].

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 the code queries that realize the mapping between the features and the completion code. For more information on code transformations, forward, and round-trip engineering using FSMLs, which are outside of the scope of this paper, we refer the reader to the related Ph.D. thesis [4]. The thesis also provides more detail

on the *generic FSML infrastructure* upon which the languages are built, and on the methodology used for developing FSMLs. Each language is specified declaratively and its metamodel is interpreted by the algorithms implemented in the infrastructure. The infrastructure also supports pluggable *mapping interpreters* which execute code queries and transformations for the mapping types they support. To date, we developed three mapping interpreters for Java, XML, and Eclipse plugin.xml files; in this paper we only focus on mapping types and code queries for Java.

4 Setup of the study

We conducted the study in three phases. The purpose of the first phase was to identify the types of structural and behavioural 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 purpose of the third phase was to implement the proposed (refined) code queries and evaluate their precision and recall on a larger number of applications.

4.1 Setup of Phase 1

The inputs to the first phase of the study are three sample FSMLs, one for each of the following frameworks: Java Applet [31], Apache Struts [9], and a part of Eclipse Workbench [27] (detailed descriptions of the languages can be found elsewhere [4]). 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 an XML configuration file. Eclipse Workbench Part Interaction (WPI) FSML captures the concepts of *editor*, *view*, *selection provider*, *selection listener*, *part listener*, *adapter provider*, and *adapter requestor*. WPI FSML has 52 features and it 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 only considered features related to Java code and omitted (i) features related to XML configuration files and (ii) features that represent referential integrity constraints, which are realized by model queries. The features related to XML configuration files in our FSMLs simply correspond to XML elements and attributes and can be retrieved with 100% precision and recall. Model queries operate on the already retrieved features and thus are irrelevant with respect to code querying. The identified types of code patterns and the implemented queries are presented in Section 5.

4.2 Setup of Phase 2

In the second phase of the study we used the prototype implementations of the three FSMLs to automatically reverse engineer a number of sample applications. The prototypes implement code queries that realize mapping definitions of the FSMLs. The unit of analysis was a *project*: an entity that groups all source artifacts of the analyzed application. For the Java Applet framework, sample applets were grouped into two projects, one with 20 examples provided by Sun and one with 51 applets collected from the Internet. 36 of the applets from the internet were the applets used in the design fragments study [18]. The authors of that study used the search string `import java.applet.Applet -site:sun.com` and they revised the search results to select applets that meaningfully used the framework. The remaining 15 applets were not used in that study but they were collected using the same method. Each of the Struts applications, Apache Roller [8] (v.3.0), Mailreader [9] (v.1.3.8), and Cookbook [9] (v.1.3.8), constitutes a separate project. Apache Roller is a large, open-source, and widely used application implementing 58 actions and using 186 forwards. Mailreader and Cookbook are small example applications provided with the framework implementing 19 and 16 actions, respectively. For the Eclipse Workbench framework, an application is encapsulated as an Eclipse plug-in. Because Eclipse plug-ins form complex dependency graphs, it is difficult to analyze plug-ins separately. However, by analyzing a plug-in that depends on most of the other plug-ins, we can analyze all these plug-ins at once. For WPI FSML, the project consisted of the *org.eclipse.pde.ui* plug-in (v.3.2), which depends on many other ui plug-ins including[1] *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 revealed 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 6.1. The results are discussed in Sections 6.3 and 8.

4.3 Setup of Phase 3

The purpose of the third phase of the study was to provide stronger evidence supporting our findings in the initial phases. To this end, we implemented the refined code queries proposed in Phase 2, slightly extended the definitions of the three FSMLs, and repeated the evaluation on a larger set of applications. For the Applet FSML, we gathered 13 additional applets from the Internet,

[1] For brevity, we omit prefix *org.eclipse.* from the names.

three extra Struts applications (Ajax Chat, Beer4all and Pools [1]) were used in the evaluation of the Struts FSML. The three applications implement 30 actions and 87 forwards, in total. For the WPI FSML we created a new plug-in that depends on a subset of the Eclipse Europa plug-ins (Eclipse 3.3.2). This way we analyzed potential interaction that can occur among 133 editors and views, 45 parts more as compared to phase 2. We analyzed the latest available versions of the applications as of March 2008. The detailed list of applications used in Phase 3 of the study is presented in Appendix A. The results are presented and discussed in Sections 7.3 and 8.

In addition to extending the set of applications to be analyzed, we also improved the FSMLs. In particular, we made changes to accommodate (i) multiple listener registrations and deregistrations and (ii) multiple values resulting from constant propagation. We also added some new features. In total, we added 8 features to Applet FSML, 1 feature to Struts FSML, and 4 features to WPI FSML. We removed one feature from WPI FSML.

4.4 Data collection process

For any given feature, we consider the code patterns that all of its instances in the reverse-engineered project correspond to. The correspondence is specified using mapping definitions attached to the features. For a feature f let

- A_f be the number of all patterns in the code that satisfy the mapping definition and that can be determined statically,
- Q_f be the number of patterns matched by the query,
- C_f be the number of patterns that satisfy the mapping definition and are matched (correctly) by the code query,
- M_f be the number of patterns that satisfy the mapping definition and are missed by the code query (false negatives),
- I_f be the number of patterns that do not satisfy the mapping definition and are matched (incorrectly) by the code query (false positives).

Note that A_f takes into consideration only patterns that can be determined statically. For example, for method calls, A_f accounts for all method calls matching the given signature, but does not account for the possible method calls through reflection. Similarly, for the values of method call arguments (such as on lines 8 and 10 of Figure 1), A_f does not account for dynamic values, such as coming from the user of the application or an input stream. In the latter case, we did not count dynamic values as false negatives, which is reflected in the following equations, whereby the recall depends on the value of A_f.

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:

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

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

In Phase 2, we collected the data as follows. 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.

In contrast to the original code queries, the values Q_f, M_f and I_f for the proposed refined code queries were obtained manually by checking whether 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.

In Phase 3, for a feature *f*, value Q_f was also returned by the prototypes. Value I_f was obtained by manually checking every feature from the automatically retrieved model and its corresponding code and counting those features that were incorrectly identified in the code. Value M_f was obtained by manual code inspection driven by the comparison of models extracted using different analysis settings. We present the details of the study in Section 7.

5 Results of Phase 1: 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 behavioural 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, behavioural patterns consist of *shadows* [21] 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 2. The first column contains Smalltalk-like expressions that can be used to specify the patterns. Each expression represents the application of a primitive mapping type to one or more parameters. For example, the expression *c* `callsTo:` *s* `receiver:` *r* applies the mapping type named `callsTo:receiver:` to the three parameters *c*, *s*, and *r*. Note that the first parameter, *c*, appears at the very front position, which is known as the *target* position. 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

Structural Pattern Expression	**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 methods: s	matches methods with signature s that are implemented or overridden by the class c. The signature may contain * for the method name to match any method name	methods
c allMethods: s	matches methods with signature s that are implemented, overridden or inherited by the class c. The signature may contain * for the method name to match any method name	allMethods
Behavioral Pattern Expression	**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(Object o): 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 2 Types of structural and behavioral code patterns

patterns in the code that match the given pattern expression. Since structural patterns can be fully retrieved 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 behavioural 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 [23] as expressions that define patterns of run-time events. In that context, crosscutting behaviour can be applied when such patterns occur at run-time. In the context of FSMLs, pointcuts provide the exact definitions of the behavioural patterns that features correspond to. In Table 2, we use meta pointcuts parametrized with variables from the pattern expressions. The parameters and macro calls prefixed with a `$` sign are replaced by the corresponding variable value or by expanding the corresponding meta pointcut, respectively. We reuse elements of syntax of AspectJ [22] and some of its extensions, namely the Data Flow Pointcut [25] and Tracematches [3]. For example, the meta pointcut for the pattern type `callsTo` uses AspectJ's `call`, `target`, and `cflow` pointcuts. This meta pointcut also uses the helper pointcut `execs`, which is defined at bottom of Table 2. Furthermore, the meta pointcut for 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. The meta pointcut for the pattern type `before` uses the Tracematches notation to define the order in which method calls occur. Finally, we had to introduce a new primitive pointcut, namely `returnTypes`. This new pointcut captures the run-time type of the object returned by a method and is used in the meta pointcut for the pattern type `retTypes`.

Tables 3-5 present fragments of the metamodels of the three FSMLs used in the study. Each row contains a feature and its mapping definition in angle

FSML Feature <Pattern Expression>
[0..*] Applet <class>
![1] extendsApplet <assignableTo: Applet>
[0..*] showsStatus <callsReceived: showStatus(String)>
[0..1] message (String) <valueOfArg: 1>
[0..1] listensToMouse
![1] implementsMouseListener <assignableTo: MouseListener>
![1] registers <callsReceived: addMouseListener(IMouseListener)>
[1] deregisters <callsReceived: removeMouseListene(IMouseListener)>
[1] deregistersSameObject <argument: 1 ofCall: ../../registers sameAsArg: 1 of ..>
[1] registersBeforeDeregisters <methodCall: ../../../registers before: ../..>
[0..*] thread <field>
![1] typedThread <fieldOfType: Thread>
[1] initializesThread <assignedNew: Thread(IRunnable)>
[1] nullifiesThread <assignedNull>
[0..*] parameter <callsReceived: getParameter(String)>
[0..1] name (String) <valueOfArg: 1>
[1] providesParameterInfo <methods: getParameterInfo()>

Table 3 Fragment of the metamodel of the Applet FSML

FSML Feature <Pattern Expression>
[0..*] Action <class>
![1] extendsAction <assignableTo: Action>
[0..1] extendsDispatchAction <assignableTo: DispatchAction>
[0..*] actionMethod <methods: *(ActionMapping, ActionForm, [...], [...])>
[0..1] overridesExecute <methods: execute(ActionMapping, ActionForm, [...])>
[0..*] forwardImpl <callsTo: findForward(String)>
[1] name (String) <valueOfArg: 1>

Table 4 Fragment of the metamodel of the Struts FSML

brackets. *Pattern expression* defines the correspondence between the features and structural and behavioural patterns in the completion code. 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. Features that do not have a pattern expression are abstract and are

FSML Feature <Pattern Expression>
[0..*] Part <class>
![1] implementsIView/IEditorPart <assignableTo: IViewPart/IEditorPart concrete: true>
[0..*] SelectionProvider <class>
![1] implementsISelectionProvider <assignableTo: ISelectionProvider>
[1]registers <callsTo: setSelectionProvider(ISelectionProvider)>
[0..*] SelectionListener <class>
![1] implementsISelectionListener <assignableTo: ISelectionListener>
[0..1] globalSelectionListener <callsTo: addSelectionListener(ISelectionListener)>
[1] deregisters <callsTo: removeSelectionListener(ISelectionListener)>
[1] deregistersSameObject <argument: 1 ofCall: ../.. sameAsArg: 1 ofCall: ..>
[1] registersBeforeDeregisters <methodCall: ../../.. before: ../..>
[0..1] globalPostSelectionListener <callsTo: addPostSelectionListener(ISelectionListener)>
[1] deregisters <callsTo: removePostSelectionListener(ISelectionListener)>
[1] deregistersSameObject <argument: 1 ofCall: ../.. sameAsArg: 1 ofCall: ..>
[1] registersBeforeDeregisters <methodCall: ../../.. before: ../..>
[0..*] specificSelectionListener <callsTo: addSelectionListener(String, ISelectionListener)>
![1] registrationPartId <valueOfArg: 1>
[1] deregisters <callsTo: removeSelectionListener(String, ISelectionListener)>
[1] deregistrationPartId <valueOfArg: 1>
[1] deregistersSameObject <argument: 2 ofCall: ../.. sameAsArg: 2 ofCall: ..>
[1] registersBeforeDeregisters <methodCall: ../../.. before: ../..>
[0..*] PartListener <class>
![1] implementsIPartListener <assignableTo: IPartListener>
[1] registers <callsTo: addPartListener(IPartListener)>
[1] deregisters <callsTo: removePartListener(IPartListener)>
[1] deregistersSameObject <argument: 1 ofCall: ../../registers sameAsArg: 1 ofCall: ..>
[1] registersBeforeDeregisters <methodCall: ../../../registers before: ../..>
[0..*] AdapterProvider <class>
![1] providesAdapter <allMethods: Object getAdapter(Class)>
![1..*] adapters (String) <returnedObjectTypes>
[0..*] AdapterRequestor <class>
![1..*] requestsAdapter <callsTo: getAdapter(Class) receiver: IWorkbenchPart>
[1] adapter (String) <valueOfArg: 1>

Table 5 Fragment of the metamodel of the WPI FSML

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 required and sufficient for establishing a correspondence to a feature are specified by the feature's essential subfeatures (marked using !). Values of the parameters of the pattern expressions can be either statically provided in the metamodel or they can be retrieved from parent features during reverse engineering. The value of the first parameter is usually not specified explicitly, in which case it is determined implicitly by locating the closest parent feature with a pattern expression that matches a code element of the required type and using the matched element as the value. For example, the pattern expression `assignableTo: Applet` for the feature `extendsApplet` requires a Java class as the value of the first parameter (i.e., `c` in Table 2), 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` in Table 3 requires two method calls and uses paths "`../../registers`" and "`..`" to retrieve calls that the features `registers` and `deregisters` correspond to.

We present the metamodels for two reasons: (i) to give the reader examples of concrete pattern expressions and (ii) to help the reader understand the tables with precision and recall presented in Sections 6.1 and 7.3.

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. During reverse engineering, the mapping definitions are interpreted by first determining the mapping type that is used by analysing the pattern expression, retrieving the values of parameters, and executing an appropriate code query. For a detailed description of how the mapping definitions are interpreted by the generic FSML infrastructure during reverse engineering, we refer the reader to the related Ph.D. thesis [4, ch. 2], as the algorithms are outside the scope of this paper.

We present the code queries that approximate behavioural patterns in Tables 6-13. The code queries are defined in the Smalltalk-like notation, similar to their corresponding pattern expressions in Table 2. For each code query, we provide a description of the results obtained by statically applying the query to the code.

Queries marked with an asterisk (*) are the ones used in the prototypes in Phase 2 of the study and they will be discussed in this section. The remaining queries are the ones we proposed as query refinements in Phase 2. These queries are the hypothetical ones that would match behavioural code patterns with 100% precision and recall. They are hypothetical in the sense of assuming perfect static analyses that could infer any code property that could be statically inferred. These queries are discussed in Section 6. In Phase 2, they were only "executed" manually; obviously, such an execution represents our "best effort" and we discuss this issue in the threats to validity. However, the refined queries were implemented in Phase 3 by making specific implementa-

tion choices about the involved static analyses, which could potentially reduce their precision and recall. The implementations are presented in Section 7.

Code Query Abbrev.	**Query Expression** / **Result**
getCallsWH*	*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*
getCallsCF	*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*

Table 6 Code queries for the *callsTo* pattern type

Code Query Abbrev.	**Query Expression** / **Result**
getCallsRec*	*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*
getCallsRecTI	*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

Table 7 Code queries for the *callsRec* pattern type

Code Query Abbrev.	**Query Expression** / **Result**
getArgValLC*	*mc* getArgValLiteralConstant: *i*
	value of the i^{th} argument of the method call *mc* retrieved from a `static final` variable or a literal
getArgValCP	*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
getArgValPE	*mc* getArgValPartialEval: *i*
	set of values of the i^{th} argument of the method call *mc* retrieved using partial evaluation

Table 8 Code queries for the *argVal* pattern type

Code queries presented in Tables 6-13 can be described based on the kind of approximation they employ. We discuss their potential false positives and false negatives.

One group of queries approximate interprocedural control flow graph of an object: `getCallsWH` and `isBeforeWH`. The idea is to search in the bodies of

Code Query Abbrev.	**Query Expression** / **Result**
argIsThis*	c thisAsArgument: i ofCall: mc_1 andArg: j ofCall: mc_2
	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
argIsPrvFieldAO	c prvFieldAsArgument: i ofCall: mc_1 andArg: j ofCall: mc_2 givenCSeq: 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

Table 9 Code queries for the *argSameObj* pattern type

Code Query Abbrev.	**Query Expression** / **Result**
isBeforeWH*	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
isBeforeCF	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

Table 10 Code queries for the *before* pattern type

Code Query Abbrev.	**Query Expression** / **Result**
getRetTypesWS*	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
getRetTypesMST	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

Table 11 Code queries for the *retTypes* pattern type

Code Query Abbrev.	**Query Expression** / **Result**
getAssgnNew*	f getAssignedNew: cc
	a set of assignments to the field f with the constructor call cc

Table 12 Code queries for the *assgnNew* pattern type

the object's class and its superclasses because the code implementing the call is likely to be found there. These queries can potentially miss patterns (false

Code Query Abbrev.	Query Expression Result
getAssgnNull*	f getAssignedNull
	a set of assignments to the field f with the `null` literal

Table 13 Code queries for the *assignNull* pattern type

negatives) located in helper classes whose code is located outside the class of the object (nested and anonymous classes are included in the search). Also, the queries can incorrectly identify patterns (false positives) in the superclasses. This happens when the call resides in the bodies of methods that get overridden and which are not reached by a `super` call.

The query `isBeforeWH` relies on the information about method callback sequence of the framework. 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. This query will miss a pattern if at least one of the two method calls to be matched is in the control flow of a callback method, but not directly in the body of that method.

Another group of queries rely on static type binding information: `getCallsRec` and `getRetTypesWS`. The former uses the type binding of the receiver of a method call to determine if it matches the specified type, while the latter uses the type binding of the return statements of a method. These queries can generate false negatives when the binding points to a type that is more general than the actual type of the returned object. The query `getRetTypesWS` will also return an inappropriate type if (i) the object to be returned 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 `argIsThis` considers the arguments of two method calls as being the same object only if they employ the `this` keyword. This approximation misses all other cases where the methods are called with the same object as argument, such as when the argument is a private field of the class.

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, respectively. These queries will miss patterns when the field is assigned with a variable which had previously been assigned the result of a constructor call or `null`, respectively. In these cases, dataflow graph traversal is necessary.

6 Results of Phase 2: Evaluation of the Simple Code Queries

6.1 Precision & Recall

Tables 14, 15, and 16 present values A_f, D_f, M_f, R_f, and P_f for every code query used for the retrieval of instances of the feature f. In the columns A_f,

the number between parenthesis denotes the number of features not counted as false negatives because they are dynamic.

The column *Query Type* contains names of code queries used for retrieving patterns of the given type. The queries for structural patterns are omitted for brevity, which results in the empty cells in the tables. Obviously, these queries are precise and complete implementations of the corresponding mapping types. For behavioural patterns, we provide queries that were used by the prototypes to approximate the corresponding mapping types. If a query retrieved less than 100% of patterns, we include manually computed data for the proposed query refinements in the subsequent rows. The cases in which the refined queries were needed to match missed patterns can be visually recognized by looking at cells in the column A_f that span multiple rows (6 cases). The column, R_f, contains the recall calculated according to the equation 4. The last column, P_f, contains the precision calculated according to the equation 3. Except for three features which have a single false positive each, the precision is always 100%. In Section 6.2 we describe the refined queries and the data is discussed in Section 6.3.

6.2 The Refined Code Queries

In Tables 6-11, the queries not marked with an asterisk represent refinements over the marked queries. We present the refined code queries because we use them as a point of reference in Tables 14-16, that is, we measure false negatives with respect to the number of patterns matched by the best refined code query. In Section 4.4 we defined A_f as the number of all patterns in the code that satisfy the mapping definition and that can be determined statically, meaning the number of patterns that can be retrieved by the best available code query. Since the refined code queries were not implemented, we computed the values of manually.

The queries `getCallsCF` and `isBeforeCF` refine `getCallsWH` and `isBeforeWH`, respectively, by correctly considering the set of available methods in an object of that class and by analyzing the call graph of this object. They will therefore ignore method calls found in methods that get overridden and are not reachable, and will detect method calls in helper classes.

Some refined queries traverse the dataflow graph backwards, beginning at a particular use of a variable, to determine its most specific type. Queries `getCallsRecTI` and `getRetTypesMST` refine `getCallsRec` and `getRetTypesWS`, respectively, because using the most specific type information for the method call receiver or the return expression potentially matches additional patterns. The query `argIsPrvFieldAO` improves `argIsThis`. It determines whether the only field assignment occurred before the first method call. This 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.

The refinement of `getArgValLC` is achieved by incrementally employing more powerful static analyses. The query `getArgValCP` also considers only constants when determining the argument value, but it uses interprocedural constant propagation to match additional patterns. The query `getArgValPE` goes beyond constant propagation and uses partial evaluation to determine argument values. Partial evaluation is an optimization technique in which the program is evaluated before runtime based on the statically available values. For example, partial evaluation may perform operations such as string concatenation for statically known strings, loop unrolling for loops with statically known bounds, and retrieving values from static array initializers.

6.3 Interpretation of the data

We discuss the data presented in Tables 14-16, each table separately. We focus on the highlighted cells. Surprisingly, for all features except three, and for

FSML Feature	**Query Type**	A_f	Q_f	M_f	R_f	P_f
[0..*] Applet		71	71	0	100	100
![1] extendsApplet		71	71	0	100	100
[0..*] showsStatus	getCallsRec	39	39	0	100	100
[0..1] message	getArgValLC	23(16)	17	6	73.91	100
	getArgValCP		18	5	78.26	100
	getArgValPE		23	0	100	100
[0..1] listensToMouse		23	23	0	100	100
![1] implementsMouseListener		23	23	0	100	100
![1] registers	getCallsRec	23	23	0	100	100
[1] deregisters	getCallsRec	10	10	0	100	100
[1] deregistersSameObject	argIsThis	10	10	0	100	100
[1] registersBeforeDeregisters	isBeforeWH	10	10	0	100	100
[0..*] thread		32	32	0	100	100
![1] typedThread		32	32	0	100	100
[1] initializesThread	getAssgnNew	30	30	0	100	100
[1] nullifiesThread	getAssgnNull	17	17	0	100	100
[0..*] parameter	getCallsRec	153	148	5	96.73	100
	getCallsRecTI		153	0	100	100
[0..1] name	getArgValLC	217(11)	132	85	60.82	100
	getArgValCP		196	21	90.32	100
	getArgValPE		217	0	100	100

Table 14 Statistics for framework-specific models retrieved using Applet FSML

the code queries used in the prototypes the precision turned out to be 100%. Precision is influenced by the number of false positives. By checking all features in the retrieved models we concluded that only three were false positives. As discussed in Section 5, all of the code queries can potentially return false positives. Therefore, finding only three false positives in the models for the analyzed applications only means that these particular applications were written

FSML Feature	Query Type	A_f	Q_f	M_f	R_f	P_f
[0..*] Action		93	93	0	100	100
![1] extendsAction		93	93	0	100	100
[0..1]extendsDispatchAction		47	47	0	100	100
[0..*] actionMethod		124	124	0	100	100
[0..1]overridesExecute		42	42	0	100	100
[0..*] forwardImpl	getCallsWH	212	212	0	100	100
[1] name	getArgValLC	211(1)	211	0	100	100

Table 15 Statistics for framework-specific models retrieved using Struts FSML

FSML Feature	Query Type	A_f	Q_f	M_f	R_f	P_f
[0..*] Part		88	88	0	100	100
![1] implementsIView/IEditorPart		88	88	0	100	100
[0..*] SelectionProvider		1	1	0	100	100
![1] implementsISelectionProvider		1	1	0	100	100
[1] registers	getCallsWH	1	1	0	100	100
[0..*] SelectionListener		8	8	0	100	100
![1] implementsISelectionListener		8	8	0	100	100
[0..1] globalSelectionListener	getCallsWH	1	1	0	100	100
[1] deregisters	getCallsWH	1	1	0	100	100
[1] deregistersSameObject	argIsThis	1	1	0	100	100
[1] registersBeforeDeregisters	isBeforeWH	1	1	0	100	100
[0..1] globalPostSelectionListener	getCallsWH	6	6	0	100	100
[1] deregisters	getCallsWH	6	6	0	100	100
[1] deregistersSameObject	argIsThis	6	6	0	100	100
[1] registersBeforeDeregisters	isBeforeWH	4(2)	4	0	100	100
[0..*] specificSelectionListener	getCallsWH	1	1	0	100	100
![1] registrationPartId	getArgValLC	1	1	0	100	100
[1] deregisters	getCallsWH	1	1	0	100	100
[1] deregistrationPartId	getArgValLC	1	1	0	100	100
[1] deregistersSameObject	argIsThis	1	1	0	100	100
[1] registersBeforeDeregisters	isBeforeWH	0(1)	0	0	100	100
[0..*] PartListener		10	10	0	100	100
![1] implementsIPartListener		10	10	0	100	100
[1] registers	getCallsWH	10	10	0	100	100
[1] deregisters	getCallsWH	9	10	0	100	90
[1] deregistersSameObject	argIsThis	16	10	6	62.5	100
	argIsPrvFieldAO		16	0	100	100
[1] registersBeforeDeregisters	isBeforeWH	15	10	6	66.66	90
	isBeforeCF		15	0	100	100
[0..*] AdapterProvider		44	44	0	100	100
![1] providesAdapter		44	44	0	100	100
![1..*] adapters	getRetTypesWS	190	132	59	69.47	100
	getRetTypesMST		190	0	100	100
[0..*] AdapterRequestor		22	22	0	100	100
![1..*] requestsAdapter	getCallsWH	68	69	0	100	98
[1] adapter	getArgValLC	62	62	0	100	100

Table 16 Statistics for a framework-specific model retrieved using WPI FSML

in a way that the queries did not return many false positives. In the following description we comment on the highlighted cells of Tables 14-16.

Table 14. *The feature* `message`. The six 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). For 16 method calls, values of arguments were not retrieved because they cannot be determined statically. We did not count these values as false negatives.

The features `deregistersSameObject` and `registersBeforeDeregisters`. In all 10 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` and `getAssgnNull`, 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 five 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, the query `getCallsRecTI` would infer that the applet is, in fact, the receiver of the five method calls.

The feature `name`. The 85 missed parameter names can be retrieved using constant propagation and loop unrolling. For three instances of the 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 two method calls was necessary to reach the static values. Therefore using the query `getArgValCP` reduced the number of false negatives to 21. Using the query `getArgValPE` further eliminates all remaining false negatives as follows. For two instances of the feature `parameter`, a call to `getParameter` was placed in a loop with a statically known loop count. For the first instance, 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 four values. For the second instance, 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 15. *The feature* `name`. In the three sample 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 16. *The concept* `SelectionListener`. The eight 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.

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 three patterns not matched by the query (2 for post selection listeners and one 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 feature* `deregisters`. The query matched one pattern more than there actually are in the control flow of the part. This is the one false positive because the matched method call resided in an overridden method which was not called using `super` and dynamic method dispatch always invokes the overriding method instead.

The features `deregistersSameObject` and `registersBeforeDeregisters`. All part listeners inherit behaviour 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. Except for one case, both calls are not false positives because all part listeners delegate to `super` in `createPartControl` and `dispose` methods, which they override. However, because one deregistration method call is a false positive, this translates to a false positive for the query `isBeforeWH` because it employs the same approximation as the query `getCallsWH`.

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: 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 seven 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% for `getArgValueLC`, 82% for `argIsThis`, 76% for `isBeforeWH`, and 98% for `getCallsRec`, which shows that even such simple queries provide very high recall. An exception here is the query `getRetTypesWS` with recall 69%. 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. The weighted average precision for queries that incorrectly identified patterns is 96% for `isBeforeWH` and 99% for `getCallsWH`.

6.4 Conclusion for Phases 1 and 2

In summary, in the first two phases of the study we identified the types of structural and behavioural patterns that the features of the three exemplar FSMLs correspond to and we provided a precise definition of behavioural patterns using meta pointcuts. We showed how knowledge about a framework, such as the order of callbacks, can be leveraged for the retrieval of behavioural patterns, such as `callsTo`, `before`, and `argSameObj`, which is undecidable in the general case. 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-program analysis. The average recall for all simple queries for behavioural patterns was 88% and the precision was 99%. We proposed refined versions of the code queries that would achieve 100% precision and recall for the sample applications (cf. Tables 6-11, queries not marked with '*'). We also observed that the results are dependent on the fact that the application code was written in a simple form, often simulating framework-provided examples by following the *Monkey see, monkey do* rule [19]. Consequently, we concluded that analyzing a larger sample of applications was needed in order to see whether this observation would hold more generally or whether it was just a property of the code we analyzed so far.

7 Results of Phase 3: Implementation and Evaluation of the Refined Code Queries

7.1 Static analysis services used by the refined code queries

The implementation of the refined code queries, as recommended in the previous section, requires four basic services: *type hierarchy*, *call graph*, *dataflow graph*, and *most-specific-type inference*. As argued in Section 2, building complete type hierarchy, call graph, and dataflow graph for framework-based applications is infeasible and, as shown in Section 6, not necessary. Therefore, we implemented these services in an incremental and on-demand form that allows the code queries to obtain the necessary information as they execute. This way, and by focusing on the framework boundary, we avoid the complete analysis of both the application and the framework and yet we are still able to retrieve patterns with higher precision and recall than in Phase 2.

7.1.1 Type hierarchy

Type hierarchy is the most basic service that is used by code queries and other services. In our implementation, the service is provided by an *incremental type hierarchy manager*, which manages a single and shared type hierarchy cache. The manager provides the following query functions:

- supertype and subtype hierarchy computation and traversal for a type,
- improved support for nested and anonymous classes as compared to the default JDT's implementation, and
- checking whether a given type is assignment compatible with another type.

7.1.2 Call graph

The call graph service is provided by a configurable *incremental call graph manager*. Its design has been influenced by the observation, confirmed by the results of the second phase of the study, that method calls related to the given class usually reside in the bodies of classes within the supertype hierarchy of the given class, including nested and anonymous classes.

The query functions provided by the call graph manager are context sensitive in order to more precisely support the analysis of dynamic (polymorphic) method calls. We refer to the class for which the analysis is performed as the *context class*. After adding the context class to the call graph manager, the manager builds an explicit call graph starting from all declared and inherited methods of the context class, which we refer to as *available methods*. Edges are created for each method or constructor call in the available methods. Therefore, the analysis is *control-flow-insensitive*.

Based on the precomputed call graph, the manager provides the following query functions:

- determining the method in the hierarchy that will be executed when a dynamic method call targets the context class,

- checking whether there exists a path between two given methods in the call graph,
- checking whether a given method is reachable from any available method of a given context class, and
- retrieving all possible implementations of a given method in the hierarchy of the context class.

The call graph manager supports different modes of operation, which are configured by flags set before the analysis. The different modes influence the size and the precision of the call graph and the time required to query it.

- The flag `hierarchical`. This flag specifies that all method calls whose targets can be statically determined are included as edges in the graph. This set includes calls to static methods, even if outside the hierarchy of the context class, and calls to super and constructor calls. Moreover, the call graph includes dynamic calls within the hierarchy of the context class since the exact target method can be determined given a context type. Dynamic calls outside the hierarchy are ignored and, therefore, this call graph does not produce any false positives.
- The flag `precise`. A superset of the hierarchical call graph, the precise call graph additionally includes dynamic calls to methods outside the context hierarchy, provided that there is a single concrete implementation of the target method in the system. This flag is the default used in all experiments in Phase 3. This call graph also yields no false positives.
- The flag `full`. This flag augments the precise call graph by including an edge in the graph for each possible implementation of the target method of a dynamic call outside the context hierarchy. Since all possible paths in the system are included this graph has no false negatives, but will potentially include false positives.

The mode severely impacts (i) the call graph's size and (ii) the precision, recall, and efficiency of the queries to the call graph. The hierarchical call graph limits the path search scope to the context hierarchy and can therefore answer queries much faster than the full call graph. The precise call graph increases the precision at the cost of efficiency. The full call graph excels in recall, but at the price of less precision and efficiency.

7.1.3 Dataflow graph

The dataflow graph service is implemented as a set of recursive query functions and is used by the code queries for the *argVal* pattern type. In other words, the dataflow graph is not represented explicitly as the call graph, but rather exists as traversals implemented by the functions. The objective of the *argVal* pattern type is to match all static values of an argument of a method call. During static analysis, we can only match the *potential* values of the given expression since there may be many execution paths at run-time. Consequently, the dataflow service is *control flow-insensitive*: when looking for static values, we consider variable initializers and all assignments, even if the assignments

are in alternative control flows. We also do not perform reachability analysis. As a result, the possible false positives include (i) values used in variable initializers, but overridden by an assignment prior to the use of the variable; (ii) values used in assignments in unreachable code; and (iii) values assigned after the use of the variable. We, however, think that those situations are rare in real code. We provide some evidence supporting that claim in the next section.

The analysis is implemented as a set of query functions, each for a different kind of code elements. We refer to the function that processes expressions as the *base function*. The functions delegate to each other, which implements the traversal. The functions return a set of values. The analysis proceeds as follows, depending on the kind of the code element being processed.

- The base function, which processes an expression, returns a static value if the expression is a literal or a final variable. Otherwise, the function delegates to an appropriate function depending on the kind of the expression.
- The function for a variable uses the base function to process the expressions that are used in the variable's initializer and at the right hand side of all assignments to that variable.
- The function for a conditional expression in the form `a ? b : c` uses the base function to process the expressions `b` and `c`.
- The function for a parenthetical expression in the form `(a)` uses the base function to process the expression `a`.
- The function for a cast expression in the form `(T) a` uses the base function to process the expression `a`.
- The function for a method or constructor parameter uses the base function to process the expressions used as the right hand side of all assignments to the parameter in the body of the method. Next, the function locates calls to the method or the constructor and uses the base function to process the expressions of arguments of the calls that correspond to the analyzed parameter.
- The function for an array access uses the base function to process expressions in the cells of the array's initializer. If an index used in the array access is unknown statically, the function processes all cells of the given array dimension.
- The function for a method call first resolves the target method declaration using the call graph manager and delegates to the following function to process the method declaration.
- The function for a method declaration uses the base function to process expressions used in all return statements.

7.1.4 Most-specific-type inference

The most-specific-type inference service is implemented as a set of recursive query functions and is used by the code queries for the `retTypes` pattern type. The service is also used by the code queries for the `callsTo` and `callsReceived` to infer the type of the method call's receiver. The service is implemented similarly to the dataflow graph service.

Depending on the kind of the code element, the analysis proceeds as follows. Again, we refer to the function that processes expressions as the *base function*. Each of the functions below returns a set of most-specific types. The types do not have to be assignment compatible to each other; the returned types only have to be assignment compatible with the static type of the code element passed to the function.

- The base function, which processes an expression, returns the most-specific type if the expression is a class instance creation. If the class instance creation statement creates an anonymous subclass, the supertype is returned. Otherwise, the function delegates to an appropriate function depending on the kind of the expression.
- The function for a variable uses the base function to process the expressions which are used in the variable's initializer and the right hand side of all assignments.
- The function for a conditional expression in the form `a ? b : c` uses the base function to process the expressions `b` and `c`.
- The function for a parenthetical expression in the form `(a)` uses the base function to process the expression `a`.
- The function for a cast expression in the form `(T) a` uses the base function to process the expression `a`; returns `T` if the base function returned a more general type then `T` for the expression `a`.
- The function for a method or constructor parameter uses the base function to process the expressions used as the right hand side of all assignments to the parameter in the body of the method. Next, the function locates calls to the method or the constructor and uses the base function to process the expressions of arguments of the calls that correspond to the analyzed parameter.
- The function for a method call first resolves the target method declaration using the call graph manager and delegates to the next function to process the declaration
- The function for a method declaration uses the base function to process expressions used in all return statements; the function returns the most specific types of all returned expressions.

7.2 Implementation of the refined code queries

In this section we briefly describe how the code queries use the services presented in the previous section.

`getCallsCF`. First, the query adds the context class to the call graph manager, which will build its call graph and connect it to the already existing call graphs in the system. The call graph manager ensures that the call graph is built at most once for the given class. The query subsequently locates all method calls of the given signature in a certain configurable search scope (detailed below). The query then needs to eliminate method calls that are not in the control flow of the context class. If the query has been configured with a

given receiver type, it starts by using the most-specific-type inference services to eliminate calls whose receiver is not assignable to the required receiver's type. Finally, it uses the call graph manager to eliminate calls that are declared in methods that are not reachable from the context class.

The search scope determines the places in the system to search for method calls and it is controlled by a flag.

- The flag `hierarchyUnits` specifies that the search scope consists of all classes in the superclass hierarchy of the context class, including the context class, and all other types declared in compilation units of these classes. This is the default setting we used in all experiments in Phase 3.
- The flag `project` specifies that the search scope consists of all classes in the class path of the analyzed project. In Eclipse, this also includes classes that belong to all plug-ins from the dependencies of the analyzed plug-in. We used this flag during the validation to locate false negatives, that is, method calls that are in the control flow of the context class but are located outside of its superclass hierarchy.

Using the `hierarchyUnits` search scope significantly reduces the number of method calls that need to be checked for reachability from the context class. However, it introduces false negatives, such as method calls located in helper or utility classes. Using the project search scope ensures that all potentially reachable method invocations are checked but it incurs a significant penalty in the time of analysis.

`getCallsRecTI`. The query first locates all method calls of the given signature in the entire project. Next, the query eliminates method calls whose receivers are not assignable to the context class, using the type hierarchy and the most-specific-type inference services.

`getArgValCP & getArgValPE`. The query `getArgValCP` uses the dataflow graph service to retrieve all expressions with static values that can potentially be the values of the given argument of the method call. We do not perform partial evaluation. In Section 6.3 when explaining the false negatives for the features `message` and `name` from Table 14, we observed that string concatenation and loop unrolling were needed to retrieve certain values (e.g., `A[i]`, where `A` is an array and `i` is a *for loop* variable). Loop unrolling is no longer necessary because the dataflow graph service is capable of analyzing arrays and retrieving the entire static content of the array from its initializer, even if the index variable of an array access is not static. Cases where some evaluation of expressions, such as, string concatenation and appending an integer to a string is necessary (e.g., expression `"arg" + i`, where `i` is an integer), remain false negatives because we did not implement partial evaluation required to compute such values.

`argIsPrvFieldAO`. In Phase 2, we proposed the query `argIsPrvFieldAO`, which returns true if both arguments of the two method calls are either `this` literals resolving to the same type or the same private field that is assigned only once before both calls. In Phase 3, we decided to relax the requirement

for the field to be private. We implemented a new query `argIsSameObject` which returns true in the following cases:

- if both arguments are `this` literals resolving to the same type. The type resolution is context sensitive, that is, if a literal resides in the context class or its supertype, the type is resolved to the context class. Otherwise, the static type binding is used.
- if both arguments are references to the same field regardless of the number of assignments. No check is performed as to the precedence of assignment with respect to the calls.

Both queries can return true with full confidence only for the `this` literals and for a field if it is private and assigned only in the initializer. In other cases, the query will still return true, and help the user of the model to understand the code by providing information about all assigned expressions.

`isBeforeCF`. The query first checks whether the method calls are in the control flow of the constructors and methods from the callback sequence. The query returns true in three cases:

- if the first method call is in the control flow of a constructor and the second is not,
- both method calls are in the control flow of methods from the callback sequence and the method of the first call is before the method of the second call in the sequence, and
- both method calls reside in the same block and the first one occurs lexically before the second one.

Note that a false positive is possible when both method calls reside directly in the `then` and `else` branches of an `if` statement, respectively. In this case the query returns false to avoid the false positive. In all other cases the query returns false.

`getRetTypesMST`. The query uses the most-specific-type inference service to retrieve most-specific types of expressions used in return statements of the given method. The query resolves the type of a returned `this` literal to the type of the context class if the literal resides in the superclass hierarchy of the context class, including the context class. Otherwise, `this` literals are resolved to the containing classes, that is, the static type binding is used.

7.3 Precision & Recall Data and Interpretation

This section presents the recall data for the refined code queries. The precision was always 100% and we did not include it in the tables. We provide the interpretation for the highlighted cells of Tables 17-19.

Table 17 contains data for the models retrieved using the Applet FSML. We extended the FSML as follows. We added two new types of listeners: key and mouse motion listeners. We accounted for the possibility of registering multiple listeners of the same type. We changed the cardinality of the features

FSML Feature	Query Type	A_f	Q_f	M_f	R_f
[0..*] Applet		84	84	0	100
![1] extendsApplet		84	84	0	100
[0..*] showsStatus	getCallsRecTI	39	39	0	100
[0..*] message	getArgValCP	31(8)	23	8	74.19
[0..*] registersKeyListener	getCallsCF	4	4	0	100
[0..*] registersMouseListener	getCallsCF	25	23	2	92
[0..*] deregisters	getCallsCF	10	10	0	100
[1] deregistersSameObject	argIsSameObject	10	10	0	100
[1] registersBeforeDeregisters	isBeforeCF	10	10	0	100
[0..*] registersMouseMotionListener	getCallsCF	15	15	0	100
[0..*] deregisters	getCallsCF	6	6	0	100
[1] deregistersSameObject	argIsSameObject	6	6	0	100
[1] registersBeforeDeregisters	isBeforeCF	6	6	0	100
[0..*] thread		34	34	0	100
[1] initializesThread <1-1>		32	32	0	100
[0..1] initializesThreadWithRunnable	getAssgnNew	27	27	0	100
[0..1] initializesWithThreadSubclass	getAssgnNew	5	5	0	100
[1] overridesRun		5	5	0	100
[1] nullifiesThread	getAssgnNull	19	19	0	100
[0..*] parameter	getCallsRecTI	173	173	0	100
[0..*] name	getArgValCP	263(14)	259	4	98.48

Table 17 Statistics for framework-specific models retrieved using Applet FSML

`message` and `name` to [0..*] since constant propagation may return multiple values for the method call arguments. We added support for recognizing an alternative way of defining threads, which is by subclassing the `Thread` class. In this case, the method `run` must be overridden in the subclass.

The concept `Applet`. *The feature* `message`. The eight false negatives are due to string concatenation. For eight method calls the values of the argument are dynamic.

The feature `registersMouseListener`. The two missed calls reside in helper classes.

The feature `name`. The four false negatives are due to loop unrolling with string and integer concatenation (`"image" + i`, where `i` ranges from 0 to 3). Note the substantial improvement in recall: 98% as compared to 61% in Phase 2. This improvement is due to the fact that parameter names are more static because they are related to the design of an applet and therefore could be retrieved using constant propagation. In contrast, status messages are more dynamic as they report to the user some events related to the execution of the applet.

Table 18 contains data for the models retrieved using the Struts FSML. We extended the FSML by adding support for *input forwards*, i.e., forwards that can be used to navigate to the forms that provided input for actions. Input forwards can be used, for example, for returning the user to the form if the form data was incorrect.

The concept `Action`. *The feature* `forward`. The nine missed method calls reside is in the Struts method `getInputForward()`, which provides the input forward retrieval service. The method first retrieves a name of the forward

FSML Feature	Query Type	A_f	Q_f	M_f	R_f
[0..*] Action		163	163	0	100
![1] extendsAction		163	163	0	100
[0..1]extendsDispatchAction		79	79	0	100
[0..*] actionMethod		181	181	0	100
[0..1]overridesExecute		115	115	0	100
[0..*] forward	getCallsCF	376	367	9	97.61
[1] name	getArgValCP	363(13)	363	0	100
[0..*] inputForward	getCallsCF	9	9	0	100

Table 18 Statistics for framework-specific models retrieved using Struts FSML

from the XML configuration file. Next, the method uses standard `getForward` method with the given name. This case well illustrates a flaw in the definition of the `callsTo` pattern type. The pattern expression matches all method calls of the given signature in the control flow of the context class regardless of whether the method calls reside in the application code or in the framework code. Clearly, in this example, it is not an action which is using the find forward service, but it is the framework using its own service to provide the input forward retrieval service. The pattern type should specify that method calls residing in the framework code should not be matched. Therefore, the nine missed calls should not be matched for the feature `forward` (note that they are not false positives either because they match the pattern expression).

The feature `name`. The query matched all statically available values. This confirms the result from Phase 2. However, in nine cases (arguments of nine calls to `getForward` in `getInputForward` method), the values of arguments were dynamic since they came from the XML configuration file. In four cases, the values of arguments were dynamic. We did not count these 13 cases as false negatives.

Table 19 contains data for the models retrieved using the Workbench Part Interactions FSML. We extended the FSML as follows. We removed the requirement for the providers and listeners to directly implement the interfaces (e.g., `ISelectionProvider`, `IPartListener`). This way workbench parts can register other classes as providers or listeners. We accounted for the possibility of multiple registrations and deregistrations. We also added a new type of part listeners.

All missed method calls reside in helper classes. See features `registers`, `deregisters`, `globalSelectionListener`, `partListener`, `partListener2`, and `requestsAdapter`.

The concept `SelectionProvider`. 83 workbench parts register a selection provider as compared to one in Phase 2. The difference is due to the relaxation of the requirement that the part has to implement the `ISelectionProvider` interface.

The concept `SelectionListener`. The feature `registersBeforeDeregisters`, a subfeature of the feature `globalPostSelectionListener`. In three cases, the order cannot be statically determined as it depends on user's actions.

The feature `registersBeforeDeregisters`, a subfeature of the feature `specificSelectionListener`. In two cases, the order cannot be statically determined as the registration depends on user's actions; they are not false negatives.

The concept `PartListener`. The feature `partListener` and its subfeature `deregisters`. In one case, the method containing a deregistration call gets overridden (not a false negative). In two cases, the deregistration method calls are not in the control flow of the part (not included in A_f for the feature `deregisters`).

The feature `registersBeforeDeregisters`, a subfeature of `partListener2`. In four cases, the order cannot be statically determined. In one case, the deregistration call is not in the control flow of a callback method because the callback method is overridden and not called using `super`. In one case, the deregistra-

FSML Feature	**Query Type**	A_f	Q_f	M_f	R_f
[0..*] Part		133	133	0	100
![1] implementsIView/IEditorPart		133	133	0	100
[0..*] SelectionProvider		83	82	1	98.80
![1..*] registers	getCallsCF	96	95	1	98.96
[0..*] SelectionListener		19	18	1	94.74
![1] registersA <1-*>					
[0..*] globalSelectionListener	getCallsCF	8	7	1	87.50
[1..*] deregisters	getCallsCF	7	7	0	100
![1] deregistersSameObject	argIsSameObject	7	7	0	100
[1] registersBeforeDeregisters	isBeforeCF	7	7	0	100
[0..*] globalPostSelectionListener	getCallsCF	10	10	0	100
[1..*] deregisters	getCallsCF	10	10	0	100
![1] deregistersSameObject	argIsThis	10	10	0	100
[1] registersBeforeDeregisters	isBeforeCF	7(3)	7	0	100
[0..*] specificSelectionListener	getCallsCF	1	1	0	100
![1] registrationPartId	getArgValCP	1	1	0	100
[1..*] deregisters	getCallsCF	2	2	0	100
[1] deregistrationPartId	getArgValCP	2	2	0	100
![1] deregistersSameObject	argIsSameObject	2	2	0	100
[1] registersBeforeDeregisters	isBeforeCF	0(2)	0	0	100
[0..*] PartListener		63	63	0	100
![1] registersA <1-*>					
[0..*] partListener	getCallsCF	71	50	21	70.42
[1..*] deregisters	getCallsCF	68	49	19	72.06
![1] deregistersSameObject	argIsSameObject	49	49	0	100
[1] registersBeforeDeregisters	isBeforeCF	48	47	1	97.87
[0..*] partListener2	getCallsCF	26	26	0	100
[1..*] deregisters	getCallsCF	29	29	0	100
![1] deregistersSameObject	argIsSameObject	29	29	0	100
[1] registersBeforeDeregisters	isBeforeCF	23(4)	23	0	100
[0..*] AdapterProvider		68	68	0	100
![1] providesAdapter		68	68	0	100
![1..*] adapters	getRetTypesMST	550(67)	548	2	99.64
[0..*] AdapterRequestor		35	22	13	62.86
![1..*] requestsAdapter	getCallsCF	141	119	22	84.40
[1] adapter	getArgValCP	119	119	0	100

Table 19 Statistics for a framework-specific model retrieved using WPI FSML

tion occurs before registration lexically in the same block. In the last case, the query correctly returns false and therefore one feature instance less is present in the model. These cases are not false negatives.

The concept `AdapterProvider`. The feature `adapters`. In two cases, the query returned a more general type than the most-specific-type. In 67 cases the type of the returned object could not be determined statically; we did not count these cases as false negatives. Most of them were inherited from framework classes: 48 from `WorkbenchPart`, 10 from `PageBookView`.

The concept `AdapterRequestor`. The feature `requestsAdapter`. 19 parts inherit six adapter request method calls from `AbstractTextEditor` (114 instances); the remaining five requests are different. One missed method call resides in a static method of a utility class and is used by 16 parts. For the remaining six false negatives, helper classes are requesting an adapter. Note that since 22 calls are missed and the feature `requestsAdapter` is an essential feature of the concept, 13 instances of the concept were also missed (3 out of the 16 parts that use the utility class also request another adapter and therefore are present).

The feature `adapter`. We did not count values of arguments of the 22 missed method calls as false negatives because the query was not executed for these method calls (cf. the feature `requestsAdapter`). In all method calls the argument was a type literal in the form `X.class`, where X is the type name.

7.4 Conclusion for Phase 3

In Phase 3 we implemented the refined code queries and evaluated their effectiveness in terms of precision and recall. In Table 20 we provide execution times and memory consumption for different settings of the search scope and call graph type. We performed all measurements on an IBM Thinkpad with one Pentium M 1800Mhz and 2Gb RAM running Windows XP. Measurements marked with a star (*) were taken on a workstation with four processors Xeon 2800Mhz, 2Gb RAM, on Ubuntu Linux 7.1. We used the second machine because we were able to allocate 1500Mb of heap as compared to the maximum of 1390Mb on the first machine. Note that the number of processors does not deeply influence performance since our analysis is single threaded and does not take full advantage of multiple processors. The columns *Applet*, *Struts*, and *WPI* contain execution times for analysis settings specified in the column *Search scope/call graph*. For WPI FSML, the numbers of features and amount of memory used are also provided. The highlighted row provides values corresponding to the data presented in Section 7.3. The values in parentheses in the column features indicate number of false negatives eliminated when using a particular setting as compared to the highlighted row. The ± sign indicates that the exact breakout into false negatives and false positives is unknown. The column *memory* contains maximum amounts of memory (in megabytes) allocated during the analysis using WPI FSML. We do not provide memory usage for Applet and Struts FSMLs as the differences are not significant.

Search scope/call graph	Applets	Struts	WPI		
	time	time	time	features	memory
hierarchyUnits/hierarchical	62s	40s	174s	1842(-98)	600 Mb
hierarchyUnits/precise	205s	61s	484s	1940	900 Mb
project/precise	318s(+2)	67s(+9)	6465s	2004(+64)	1270 Mb
project/full	394s(+2)	71s(+9)	29468s	7560(±5620)	1380 Mb
			14848s*	7560(±5620)	1390 Mb*

Table 20 Time and memory statistics for various analysis settings. The highlighted row contains values corresponding to the data presented in Section 7.3.

The search scope has the biggest influence on the number of missed method calls as all of the missed calls resided in utility or helper classes. However, using the project as a search scope significantly increases analysis time in WPI because of the huge number of method calls that need to be checked for reachability from each context class. For example, in our installation of Eclipse there are 61 selection provider registrations, 61 selection listener registrations and deregistrations, 188 part listener registrations and deregistrations, and 985 adapter requests. In total, there are 1295 method calls that need to be checked for reachability for each of the 133 workbench parts in the analyzed project. Despite the significant cost of checking all method calls in the entire project, only 57 more features related to method calls were retrieved.

For the WPI FSML, the analysis using the `project/full` settings retrieved 5620 additional instances of features as compared to the highlighted row. The majority of feature instances were false positives; however, we have verified, that 64 of those were not false positives. We verified only some of the new feature instances at random and all of them were false positives. As a comparison, a model retrieved using the `project/precise` settings contained 1940 feature instances and a model retrieved using the `project/full` settings contained 7560 feature instances. We can conclude that, in the case of WPI FSML, using the full call graph is not feasible due to the large number of false positives and long analysis time.

As described in Section 7.1, the query `getArgValCP` can have false positives in certain cases, such as when a variable is assigned after it is used or a value of an initializer is always overridden by an assignment. Those false positives are possible since the dataflow graph service is flow insensitive. However, after verifying the retrieved models we concluded that those cases were not found in the analyzed code and the query did not have any false positives.

Similarly, none of the other queries returned false positives. The query `getCallsCF` eliminated potential false positives by checking reachability. The query `argIsSameObject` matched even if the number of assignments was greater than one and also matched for public or protected fields. There was always a single assignment with an object and the additional assignments, if any, were always assigning the `null` literal. The public and protected fields were only initialized and never assigned.

In Section 7.3, we observed a flaw in the definition of the `callsTo` pattern type, whereby all method calls satisfying a pattern expression are matched,

regardless of their location. As frameworks often use their own services to provide other services, we recommend restricting the definition of the pattern type to only match method calls to framework services which do not reside in the framework code. This way, matching the framework using its own services as being used by the application can be avoided.

Finally, the weighted average recall for the refined code queries is as follows: `getCallsCF`: 91.79%, `getCallsRecTI`: 100%, `getArgValCP`: 98.45%, `argIsSameObject`: 100%, `isBeforeCF`: 98.98%, `getRetTypesMST`: 99.64%, `getAssgnNew`: 100%, `getAssgnNull`: 100%. The weighted average recall for all the refined code queries is 96.69%.

8 Discussion

8.1 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 [24].

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 refined queries in Phase 2, which was performed by manual code inspection. If patterns were missed, the results would indicate precision and recall values greater than they really were. In Phase 2, both threats were minimized by (i) having two of the authors independently collect and compare the data and (ii) supporting the manual inspection of code with two code query tools: JQuery [14] and the built-in Eclipse Java Development Tools [16] search. In Phase 3, only one of the authors verified the data; the author additionally used the `project/precise` and `project/full` analysis settings to locate false negatives.

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, that is, the mapping types used for defining them. For example, highly dynamic concepts of frameworks such

as Java Swing prescribe the construction of complex object structures, which are difficult to analyze statically. In the third phase, we extended the FSMLs; however, the extensions did not significantly differ from the existing features because they also used the available mapping types. Therefore, we claim that only instances of the concepts whose correspondence to code patterns can be described using the mapping types presented in this paper can be extracted from the completion code with high precision and recall.

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 we obtained from the internet that meaningfully use the framework (not *toy* examples). The goal was to reduce the potential of bias that would come from using only applications that strictly follow the framework examples. In Phase 3, we analyzed more applications from different sources in order to provide more supporting evidence. It is important to note that our sample consisted of applications that use the frameworks directly. 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 would be necessary.

Another threat to external validity is related to the design of the study. An ideal design would divide the applications into two disjoint sets: a learning set and a testing set. Such a setup aims at demonstrating that the code queries designed for the applications from the learning set generalize to different applications from the testing set. However, the design of our study was different. In Phase 2, there was no explicit learning set: the code queries were designed based on our experience, API documentation, and articles with code samples, and the testing set was the initial set of applications (cf. Section 4.2). The learning set in Phase 3 was the testing set from Phase 2. The testing set in Phase 3 was a substantially extended version of the testing set from Phase 2 (cf. Section 4.3). In particular, we added new applications for all three frameworks and, additionally, we used version 3.3 of Eclipse (Europa, 2007) as compared to version 3.2 (Callisto, 2006) used in Phase 2.

The design of our experiment compares to the ideal design as follows. Assuming that the experience used to define the initial queries corresponds to some implicit learning set, Phase 2 comes close to the ideal design since the implicit learning set and the testing set were disjoint. In Phase 3, the learning and testing sets were not disjoint; however, because (i) the testing set was substantially extended, (ii) the precision was 100% for all applications in the testing set, and (iii) the recall was very high (weighted average of 96%) for these applications, we conclude that the refined code queries also worked well for the new applications in the extended testing set. Furthermore, when manually inspecting the code we did not see a significant increase of the number of false negatives in the new applications, which would have indicated that the queries worked well only for the applications that were also present in the learning

set. It is important to note that we did not look at the new applications added in Phase 3 when designing the refined code queries. Also, the precision and recall for the new features added in Phase 3 were evaluated using the extended testing set. Also note that constructing two disjoint sets of applications for the Eclipse framework is not possible because any set of applications would always have to include many required plug-ins that are the common dependencies.

8.2 Empirical approach to code 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 are queries `isArgPrvFieldAO` and `isBeforeCF`. Our study shows that devising heuristics by interpreting the data from the analysis of real applications can lead to efficient approximate code queries that still offer high precision and recall. This approach is in contrast to the general one, 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 [19]) and recent research on design fragments [18] suggest that developers commonly copy existing examples and utilize common programming micropatterns when using frameworks. Therefore, we believe, empirical query refinement can lead to efficient code queries that provide high precision and recall when applied to real code.

8.3 Difficulties of analyzing and understanding framework-based code

We encountered several challenges during the implementation of the refined code queries and the execution of the study. In this section, we briefly outline our findings.

Multiple levels of abstraction. In Eclipse, we saw that some views and editors have up to eight superclasses, sometimes two or three of them abstract. Typically, workbench parts also implement multiple interfaces. Parts inherit some behaviours and override some other behaviours. In the end, it is difficult to understand what exactly the final behaviour of the given class is. For example, in a few cases, we saw that certain classes override a method that contains a deregistration method call and never call the overridden method using `super`. Such overrides effectively remove the implementation of a mandatory feature and result in an API rule violation. The developers might not have been aware of doing so.

Also, in such deep type hierarchies, the statically known type (that is, the type binding) of the receiver of a method call is an interface or an abstract class. Such polymorphic calls are difficult to understand and analyze because any of the receiver's type subclasses could potentially be the actual receiver. Often, the actual type of the receiver cannot be statically determined.

Nested and anonymous classes. The widespread use of nested and anonymous classes in frameworks and framework-based applications poses many difficulties for the static analysis of such systems and the verification of the correctness of the analysis results.

Eclipse's `AbstractTextEditor`, for instance, contains 24 nested classes and interfaces. These nested members form inheritance hierarchies of their own, often implementing interfaces and extending primary classes, or even extending other nested classes. Since nested classes are allowed to call methods of their outer classes (provided the nested class is not static) a dynamic method call found inside a nested class must be analyzed both in terms of the nested class' own inheritance hierarchy and in terms of the hierarchy of its outer class because the outer class could be extended and the target method overridden.

Classes that contain many nested classes also become harder to understand. `AbstractTextEditor` has almost 7000 lines of code and many inheritance hierarchies in the same file. This makes it harder for developers to understand and extend the code and also hinders the manual verification of the correctness of the implemented analyses.

In Java, all nested classes, including anonymous classes, get compiled to a separated class file. While the naming scheme for regular nested classes is trivial, because they are named entities, compilers often differ in the naming scheme applied to anonymous classes. Our analysis relies on JDT to parse source or compiled code and build ASTs. If the compiler with which the anonymous class was compiled does not use the same naming scheme as JDT, it becomes impossible to analyze the code because JDT cannot find the corresponding class file.

Another difficulty we encounter relates to the life cycle of such classes. A static nested class and an anonymous class are very similar to a primary class in that they have their own life cycles. However, non-static nested classes are bound to the life cycle of their outer classes. Therefore, for all purposes, we consider nested classes as part of their outer classes. For example, a method call in the control flow of the nested class is considered to be also in the control flow of the outer class. Despite the fact that anonymous classes have their own life cycle we also consider them as part of the outer class. The rationale is that anonymous classes are usually created with the sole purpose of implementing a certain type with callback methods expected by the framework instead of implementing the type directly by the outer class. When the anonymous class executes the behaviours defined in its callback methods, we consider that it is executing *on behalf* of its outer class.

Adapter requestor/provider mechanism. The adapter requestor/provider mechanism is highly flexible as it allows for unplanned interactions in a dynamic platform such as Eclipse. Adapter providers have no knowledge of who is using the provided adapters. Also, adapter requestors do not need to know the details about the provider and can rely solely on the provided adapter. However, understanding the interactions of this type is non-trivial. The WPI FSML is an attempt to match the providers with requestors and present the result in the form of a model to help with understanding of this

highly dynamic aspect of the code. Our results show that such interactions can be discovered automatically with high precision and recall.

9 Related and Future Work

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

Leveraging domain knowledge in program understanding. The idea of using domain knowledge in program understanding is not new. DeBaud et al. presented two case studies in 1994 that explore the relationship between domain analysis and reverse engineering [15]. The first study uses an object-oriented framework as a source of the domain knowledge; that is "the domain description can give the reverse engineer a set of expected constructs to look for in the code". The studies were performed manually by the authors. In comparison, in FSMLs the domain knowledge is embodied in the metamodel. The metamodel is directly interptered during reverse engineering, which is an automatic process.

Later, Rugaber reported on a variety of case studies conducted to evaluate different domain knowledge representation approaches in the context of program understanding: predicate logic, algebraic specifications, frame-based knowledge representation, entity-relationship models, static object models, and object-oriented frameworks [29]. In contrast, in this paper we show that representing domain knowledge as FSMLs allows performing reverse engineering automatically.

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. Current generic code query tools for Java, such as JQuery [14], JTL [11], and CodeQuest [20] cannot query for the kinds of behavioural patterns required for the retrieval of framework-specific models. In particular, the dynamic pattern types presented in Table 2 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, such as SOOT [32] usually build a complete control flow graph of the application, which is a prerequisite for many other static analyses. However, as discussed in Section 2, the analysis of framework-based applications 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 [28]

is an approach to the 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 or framework. Zhang et al. [33] propose an algorithm for computing a call graph of an application in the presence of callback methods.

It is also important to note that our code queries are capable of analysing OSGi framework's bundles. OSGi [26] is a component framework and bundles are a kind of components. Eclipse is build on top of OSGi and Eclipse plug-ins are OSGi bundles. OSGi provides an advanced dependency mechanism in which bundles can specify exact versions of other bundles they depend on. The OSGi dependency mechanism is independent of the regular Java classpath mechanism and therefore special support is required.

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 [4,5]. 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., [10]. 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 in 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 could also be used to design code queries that provide the highest precision and recall.

10 Conclusion

Framework-specific models describe how framework-provided concepts are instantiated in the application code. Automatic location of concept instances requires matching structural and behavioural patterns in the code, which can be realized by code queries. In this paper, we evaluated the precision and recall of simple and refined code queries that can be used for model retrieval. We provided evidence that fast retrieval of high-quality models from framework-based application code is feasible for concepts whose correspondence to code patterns can be defined using the presented mapping types. The average recall of the refined queries for behavioural patterns is 96% and the precision is 100% for the analysis settings we used in Phase 3. We also showed that it is possible to achieve greater recall at the expense of analysis time without sacrificing the precision.

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

References

1. Struts applications project. `http://sourceforge.net/projects/struts/`.
2. *Spring Framework Manual*, 2008. `http://www.springframework.org/`.
3. 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.
4. M. Antkiewicz. *Framework-Specific Modeling Languages*. PhD thesis, University of Waterloo, 2008. `http://hdl.handle.net/10012/4030`.
5. M. Antkiewicz, T. T. Bartolomei, and K. Czarnecki. Automatic extraction of framework-specific models from framework-based application code. In *ASE '07: Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*, pages 214–223, 2007.
6. M. Antkiewicz and K. Czarnecki. Framework-specific modeling languages with round-trip engineering. In *MODELS*, volume 4199 of *LNCS*, pages 692–706, 2006.
7. M. Antkiewicz and K. Czarnecki. Design space of heterogeneous synchronization. In *GTTSE*, 2008.
8. Apache Software Foundation. *Roller Weblogger 3.0*. `http://rollerweblogger.org/`.
9. Apache Software Foundation. *Struts User's Guide*. `http://struts.apache.org/1.3.8/index.html`.
10. 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.
11. T. Cohen, J. Y. Gil, and I. Maman. JTL: the Java tools language. In *OOPSLA'06*, pages 89–108, 2006.
12. K. Czarnecki, S. Helsen, and U. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process Improvement and Practice*, 10(1), 2005. Special issue on Software Variability: Process and Management.
13. K. Czarnecki and A. Wasowski. Feature diagrams and logics: there and back again. In *SPLC '07: Proceedings of the 11th International Software Product Line Conference (SPLC 2007)*, pages 23–34, 2007.
14. K. De Volder. JQuery: A generic code browser with a declarative configuration language. In *PADL'06*, volume 3819 of *LNCS*, pages 88–102, 2006.
15. J.-M. DeBaud, B. Moopen, and S. Rugaber. Domain analysis and reverse engineering. In *ICSM '94: Proceedings of the International Conference on Software Maintenance*, pages 326–335, 1994.
16. Eclipse Foundation. *Java Development Tools*. `http://www.eclipse.org/jdt/`.
17. EJB 3.0 Expert Group. *JSR 220: Enterprise JavaBeansTM, Version 3.0*, 2006. `http://java.sun.com/products/ejb`.
18. G. Fairbanks, D. Garlan, and W. Scherlis. Design fragments make using frameworks easier. In *OOPSLA'06*, pages 75–88, 2006.
19. E. Gamma and K. Beck. *Contributing to Eclipse: Principles, Patterns, and Plugins*. Addison-Wesley, 2003.
20. 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.
21. E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In *AOSD '04*, pages 26–35, 2004.
22. 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.
23. 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.
24. 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.

25. H. Masuhara and K. Kawauchi. Dataflow pointcut in aspect-oriented programming. In *APLAS'03*, volume 2895 of *LNCS*, pages 105–121, 2003.
26. OSGi Alliance. *OSGi Service Platform, Core Specification, Release 4, Version 4.1*, 2007. `http://www.osgi.org/Specifications`.
27. C. Pandit. *Make your Eclipse applications richer with view linking*, 2005. `http://www-128.ibm.com/developerworks/opensource/library/os-ecllink/`.
28. 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.
29. S. Rugaber. The use of domain knowledge in program understanding. *Annals of Software Engineering*, 9(1-4):143–192, 2000.
30. Sun Microsystems. *Java Server Faces*. `http://java.sun.com/javaee/javaserverfaces/`.
31. Sun Microsystems. *Java Tutorials, Lesson: Applets*. `http://java.sun.com/docs/books/tutorial/deployment/applet/index.html`.
32. R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: is it feasible? In *CC '00: Proceedings of the 9th International Conference on Compiler Construction*, pages 18–34, London, UK, 2000. Springer-Verlag.
33. W. Zhang and B. G. Ryder. Constructing accurate application call graphs for Java to model library callbacks. In *SCAM 2006*, pages 63–74, 2006.

A Applications used in the third phase

A.1 Applets

The set of applets used in the study consists of 84 applets. The applets are divided in four groups.

- 20 aplets examples shipped with Suns JDK;
- 51 applets obtained from the internet by George Fairbanks and used in his study of design fragments: ANButton, Antacross, AquaApplet, BlinkingHelloWorld2, BrokeredChat, Bsom1,ButtonTest, Client, ConsultOMatic, ContextTestExecutor, Demographics, DotProduct, Envelope, ErrorMessage, Fireworks, FormelnApplet, GammaButton, Geometry, HelloTcl, HitMeter, HmFetcher, Iagttager, InspectClient3, JScriptExample, KeyboardAndFocusDemo, LinProg, MarchingAnts, MouseDemo, MyApplet, MyApplet, NickCam, ScatterPlotApplet, Scope, SilentThreat, SimplePong, SimpleSunApplet, SmtpApp, SuperApplet, SwatchITime, hyperbolic.Test, TetrisApp, URLExampleApplet, ungrateful.Ungrateful, ungrateful.OutPanel, UrcrcCalendar, VeChat, notprolog.WPrologGUI, notprolog.WProlog, WebStart, YmpyraAppletti, CaMK;
- 8 applets by R. Bowles: Bioquiz, Calculator, Crystal, Frogs, LightRays, Mandel, Mastermind, Starscape; and
- 5 applets from three open-source project (SourceForge): JugglingLab (3 applets), snirc 1.0 (1 applet: Chat), sudoku (1 applet: Main).

A.2 Struts

The set applications used in the study consists of 6 applications.

- 2 example Struts applications shipped with the framework: Cookbook and Mailreader 1.3.8;
- 1 large, open-source, production quality application: Apache Roller Weblogger 3.1; and
- 3 small, open-source applications: Ajax Chat 1.2, Beer4all, and Pools 2.5.

A.3 Eclipse

We created a single plug-in which depends on 227 plug-ins from our custom Eclipse Europa installation. Here we list only the main features: Eclipse 3.3.2, Ant 1.7, EMF 2.3, Help 3.3.2, JDT 3.3.2, Jsch 0.1.31, PDE 3.3.2, Team 3.3.2, WST Common 2.0.2, IBM ICU 3.6.1, TeXlipse 1.2.1, JUnit 3.8.2, GEF 3.3.2, Jetty 5.1.11, Jasper 5.5.17, Lucene 1.9.1, ASTView 1.1.3, JEView 1.0.4.