Framework-Specific Modeling Languages

by

Michał Antkiewicz

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2008

© Michał Antkiewicz 2008

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Michał Antkiewicz

Abstract

Framework-specific modeling languages (FSMLs) help developers build applications based on object-oriented frameworks. FSMLs formalize abstractions and rules of the framework's application programming interfaces (APIs) and can express models of how applications use an API. Such models, referred to as *framework-specific models*, aid developers in understanding, creating, and evolving application code.

We present the concept of FSMLs, propose a way of specifying their abstract syntax and semantics, and show how such language specifications can be interpreted to provide reverse, forward, and round-trip engineering of framework-specific models and framework-based application code.

We present a method for engineering FSMLs that was extracted post-mortem from the experience of building four such languages. The method is driven by the use cases that the FSMLs under development are to support. We present the use cases, the overall process, and its instantiation for each language. The presentation focuses on providing concrete examples for engineering steps, outcomes, and challenges. It also provides strategies for making engineering decisions.

The presented method and experience are aimed at framework developers and tool builders who are interested in engineering new FSMLs. Furthermore, the method represents a necessary step in the maturation of the FSML concept. Finally, the presented work offers a concrete example of software language engineering. FSML engineering formalizes existing domain knowledge that is not present in language form and makes a strong case for the benefits of such formalization.

We evaluated the method and the exemplar languages. The evaluation is both empirical and analytical. The empirical evaluation involved measuring the precision and recall of reverse engineering and verifying the correctness or forward and round-trip engineering. The analytical evaluation focused on the generality of the method.

Acknowledgements

I would like to thank the people who have contributed to this thesis. First and foremost, I would like to thank my supervisor Professor Krzysztof Czarnecki who enormously helped to shape, reinvent, and crystallize the material presented in this thesis in countless discussions. I am deeply grateful that despite his busy schedule, Professor Czarnecki always found the time to delve into the details of my research and that he was never satisfied with "easy" answers. His emphasis on the quality of research is a valuable lesson I am striving to follow.

I would also like to thank Professor Ladan Tahvildari, Professor Kostas Kontogiannis, and Professor Michael Godfrey for serving on my thesis committee. I thank Professor Ralf Lämmel for being my external examiner.

I am also deeply grateful to my wife Agata, who's continuous support, trust, and understanding were crucial to finishing this thesis.

I would like to thank my colleagues from the Generative Software Development Lab at the University of Waterloo who contributed to this research and its implementation: Thiago Tonelli Bartolomei, Herman Lee, and Matthew Stephan. I also thank Henry Lau for his work on the implementation.

Acknowledgment of contributions

Most of the material presented in this thesis was taken from existing and submitted papers. The structure of the thesis is based on [12], to which my personal contribution was approximately 70%. Chapter 4 is based on [14], to which my personal contribution was approximately 70%. Section 2.5 is based on [10], to which my personal contribution was approximately 90%.

Michał Antkiewicz

As one of the authors of [10, 12, 14], I fully acknowledge the thesis author's statement above and give full permission to use any part of the papers we co-authored mentioned above. I also fully acknowledge that the thesis represents original research conducted by the thesis author.

Krzysztof Czarnecki

As one of the authors of [14], I fully acknowledge the thesis author's statement above and give full permission to use any part of the papers we co-authored mentioned above. I also fully acknowledge that the thesis represents original research conducted by the thesis author.

Thiago Tonelli Bartolomei

As one of the authors of [12], I fully acknowledge the thesis author's statement above and give full permission to use any part of the papers we co-authored mentioned above. I also fully acknowledge that the thesis represents original research conducted by the thesis author.

Matthew Stephan

Dedication

I would like to dedicate this thesis to my mother Małgorzata and my wife Agata.

Contents

List of Tables xii

List of Figures xiv

1 Introduction 1

1.1 Contributions . . . 5

2 Framework-Specific Modeling Languages 6

2.1 Overview . . . 6

2.2 Feature Models of Abstract Syntax . . . 10

2.3 Mapping Definitions . . . 15

2.4 Code Queries and Transformations . . . 17

2.5 Metamodel Interpretation Algorithms . . . 19

2.5.1 Forward engineering . . . 20

2.5.2 Reverse engineering . . . 21

2.5.3 Round-trip engineering . . . 24

2.6 The Generic FSML Infrastructure . . . 28

2.7 Summary . . . 29

3 Method for Engineering FSMLs 30

3.1 Overview . . . 30

3.2 Inception . . . 32

3.3 Elaboration . . . 39

3.4 Construction . . . 42

3.5 Transition . . . 49

3.6 Summary . . . 51

4 Evaluation of Reverse Engineering 52

4.1 Introduction 52
4.2 Challenges of Statically Analyzing Completion Code 53
4.3 Setup of the Study 54
4.3.1 Setup of phase 1 54
4.3.2 Setup of phase 2 55
4.3.3 Setup of phase 3 56
4.3.4 Data collection process 56
4.4 Results of Phase 1: Code Patterns & Code Queries 58
4.5 Results of Phase 2: Evaluation of the Simple Code Queries 64
4.5.1 Precision & recall 64
4.5.2 The Refined Code Queries 65
4.5.3 Interpretation of the data 66
4.5.4 Conclusion for phases 1 and 2 70
4.6 Results of Phase 3: the Refined Code Queries 70
4.6.1 Static analysis services used by the code queries 70
4.6.2 Implementation of the refined code queries 75
4.6.3 Precision & recall data and interpretation 77
4.6.4 Conclusion for phase 3 80
4.7 Discussion 82
4.7.1 Threats to validity 82
4.7.2 Empirical approach to code query refinement 84
4.7.3 Difficulties of analyzing and understanding framework-based code 84
4.8 Conclusion 86

5 Evaluation of Forward Engineering 87

5.1 Introduction 87
5.2 Code Transformations for Java 88
5.3 Setup of the Test 90
5.4 Results and Discussion 93
5.4.1 Threats to validity 94
5.5 Conclusion 95

6 **Evaluation of Round-Trip Engineering** **96**
6.1 Introduction . . . 96
6.2 Setup of the Tests . . . 97
6.3 Results and Discussion . . . 98
6.3.1 Results of test 1 . . . 98
6.3.2 Results of test 2 . . . 99
6.3.3 Threats to validity . . . 99
6.4 Conclusion . . . 101

7 **Evaluation of the Method** **102**
7.1 Evaluation of the Exemplar Languages . . . 102
7.1.1 Framework API understanding . . . 102
7.1.2 Completion code understanding and analysis . . . 103
7.1.3 Completion code generation and evolution . . . 105
7.2 Threats to Validity . . . 106

8 **Related Work** **109**
8.1 Framework Instantiation . . . 109
8.2 Reverse Engineering . . . 111
8.3 Forward Engineering . . . 112
8.4 Round-Trip Engineering . . . 114
8.5 Reengineering . . . 114
8.6 FSML Engineering Method . . . 115

9 **Conclusion** **118**
9.1 Summary of Contributions . . . 118
9.2 Limitations and Future Work . . . 119

APPENDICES **121**

A **Mapping types, Constraints, and Forward parameters** **122**

B **Applications Used in the Third Phase of the Evaluation of Reverse Engineering** **124**
B.1 Eclipse . . . 124
B.2 Struts . . . 124
B.3 Applets . . . 124

C Applications Used in the Computation of the CDC and CDO Metrics 126

C.1 Eclipse 126

C.2 Struts 126

C.3 Applets 126

D Models Used in the Evaluation of Forward and Round-trip Engineering 128

E Traces of the Execution of the Tests in the Evaluation of Round-trip Engineering 135

F Complete Metamodels of the Exemplar FSMLs 138

F.1 Metamodel of WPI FSML 139

F.2 Metamodel of Struts FSML 141

F.3 Metamodel of Applet FSML 142

F.4 Metamodel of EJB FSML 144

References 147

List of Tables

2.1 Examples of languages, their artifacts, and language definition formalisms . . . 7

2.2 Selected mapping types for structural code patterns for Java 15

2.3 Selected Mapping types for behavioural code patterns for Java . . . 17

2.4 Selected code transformations . . . 21

2.5 Selected code queries . . . 23

2.6 Synchronization States Decision Table . . . 27

3.1 Inception of the exemplar FSMLs . . . 33

3.2 Overview of the Use Cases . . . 34

3.3 Distribution of features over sources of knowledge . . . 37

3.4 Legend of Features' Sources . . . 37

4.1 Mapping types for structural code patterns for Java . . . 58

4.2 Mapping types for behavioural code patterns for Java . . . 59

4.3 Code queries for the *callsTo* mapping type . . . 62

4.4 Code queries for the *callsRec* mapping type . . . 62

4.5 Code queries for the *argVal* mapping type . . . 62

4.6 Code queries for the *argSameObj* mapping type . . . 62

4.7 Code queries for the *before* mapping type . . . 63

4.8 Code queries for the *retTypes* mapping type . . . 63

4.9 Code queries for the *assgnNew* mapping type . . . 63

4.10 Code queries for the *assignNull* mapping type . . . 63

4.11 Statistics for framework-specific models retrieved using Applet FSML 66

4.12 Statistics for framework-specific models retrieved using Struts FSML 67

4.13 Statistics for a framework-specific model retrieved using WPI FSML 68

4.14 Statistics for framework-specific models retrieved using Applet FSML 77
4.15 Statistics for framework-specific models retrieved using Struts FSML 78
4.16 Statistics for a framework-specific model retrieved using WPI FSML 79
4.17 Time and memory statistics for various analysis settings 81

5.1 Code transformation for the *class* mapping type 88
5.2 Code transformation for the *assignableTo* mapping type 89
5.3 Code transformation for the *field* mapping type 89
5.4 Code transformation for the *methods* mapping type 89
5.5 Code transformation for the *argIsThis* mapping type 89
5.6 Code transformation for the *argIsField* mapping type 89
5.7 Code transformation for the *argIsNew* mapping type 89
5.8 Code transformation for the *argIsVar* mapping type 90
5.9 Code transformation for the *callsTo* mapping type 90
5.10 Code transformation for the *callsRec* mapping type 90
5.11 Code transformation for the *argVal* mapping type 90
5.12 Code transformation for the *argSameObj* mapping type 90
5.13 Code transformation for the *retTypes* mapping type 90
5.14 Code transformation for the *assgnNew* mapping type 91
5.15 Code transformation for the *assgnNull* mapping type 91

7.1 Concern diffusion over components (CDC) 104
7.2 Concern diffusion over operations (CDO) 104

A.1 Mapping types for structural code patterns for plugin.xml 122
A.2 Mapping types for structural code patterns for XML 122
A.3 Constraints 123
A.4 Constraints and parameters for forward engineering 123

List of Figures

2.1 Overview of model-supported application engineering using FSMLs 8
2.2 Mapping definitions specify the feature-to-code-pattern correspondence 9
2.3 Fragments of the Java Applet Tutorial 10
2.4 Feature model for the concept applet 12
2.5 Feature configurations of two applet instances 14
2.6 Mapping definitions for features from the applet feature model . . . 16
2.7 Sample code described by the first configuration from Table 2.5 . . 18
2.8 Code generated for the first configuration from Figure 2.5 21
2.9 Artifacts and processes of agile round-trip engineering 24
2.10 Feature model for the concept Applet including key annotations . . 26
2.11 Architecture of the generic FSML infrastructure 28

3.1 FSML life cycle: iterations and phases 31
3.2 FSML life cycle: phases and their dominant activities 32
3.3 Excerpt of WPI FSML Metamodel 40
3.4 Excerpt of Struts FSML Metamodel 41
3.5 Excerpt of Applet FSML Metamodel 42
3.6 Excerpt of EJB FSML Metamodel 43

4.1 Fragment of the metamodel of the Applet FSML 60
4.2 Fragment of the metamodel of the Struts FSML 60
4.3 Fragment of the metamodel of the WPI FSML 61

D.1 A model and the code generated using Applet FSML (`Applet1`) . . 129
D.2 A model and the code generated using Applet FSML (`Applet2`) . . 130
D.3 A model and the code generated using WPI FSML (`View1`) 131
D.4 A model and the code generated using WPI FSML (`View2`) 131

D.5 A model and the code generated using WPI FSML (`View3`) 132
D.6 A model and the code generated using WPI FSML (`Editor1`) . . . 133
D.7 A model and the code generated using WPI FSML (`Editor2`) . . . 133
D.8 A model and the code generated using WPI FSML (`Editor3`) . . . 134

Chapter 1

Introduction

Object-oriented frameworks are widely used as a basis for the development of applications in many domains. Frameworks provide *domain-specific concepts* that are generic units of functionality. Developers create framework-based applications by writing *framework completion code* that instantiates these concepts. For example, the framework underlying Eclipse's workbench offers concepts such as views and editors. Eclipse's outline view and Java editor are specific instances of these concepts. The instantiation of the concepts require the developers to perform *implementation steps*, such as subclassing framework-defined classes, implementing framework-defined interfaces, and calling appropriate framework services. Concept instantiation is governed by the *application programming interface* (API) of the framework, which specifies the exposed programming elements, for example, the classes to subclass, the methods to call, and how they should be used.

Framework APIs are often complex and difficult to use. They may provide many *concept variants* and several alternative ways of instantiating them. For example, an Eclipse editor may be single- or multi-page. Furthermore, an action can be added to the editor's toolbar directly or by using a special action contributor mechanism. Some implementation steps are different for a multi-page editor compared to a single-page one. Furthermore, the developers have to respect API-prescribed *constraints* on the implementation steps, such as having to instantiate a multi-page action contributor for a multi-page editor rather than a regular action contributor. Finally, the developers also have to follow general *rules of API engagement*. For example, the callbacks that are called by Eclipse's User Interface (UI) framework API must not be blocking.

API documentation, code examples, and wizards, if available, offer some support in writing completion code. Cookbook-style articles and tutorials can be effective in teaching how framework-provided concepts should be instantiated. However, their main drawback is their passive nature, meaning that the developers still have to perform the necessary implementation steps manually. Since steps implementing a particular concept are often scattered across the application code and tangled with steps implementing other concepts, writing and understanding completion

code can still be challenging. This scattering and tangling is also the reason why sample applications may be difficult to use as an implementation guide. Furthermore, cookbook-style documentation is often partial and does not cover the full range of concept variants. In contrast, API reference documentation, such as that produced by JavaDoc, is usually more complete. However, this form of documentation describes only individual API elements, such as interface methods, and does not explain the higher-level concepts whose implementation involves multiple API elements. Code generation wizards, as offered by some frameworks, represent an active form of concept instantiation knowledge. Unfortunately, they usually cannot be re-run with different settings after the generated code was modified manually. Also, they typically cover only a few of the most used concept variants and they do not provide traceability between the configuration parameters and the generated code.

We believe that the existing support for the application developers can be significantly improved by taking a language-oriented perspective on framework API usage. In essence, the framework's API implicitly defines a domain-specific language in which domain-specific concepts are implemented and exposed using the mechanisms of the framework's programming language. The implementation is supplemented with descriptions in natural language whenever the programming language is insufficient. For example, the concept of a Java Applet being a mouse listener corresponds to several Java Applet API elements, including two API calls to register and then deregister a mouse listener. Whereas the API calls are represented by Java, the higher-level concept of an Applet being a mouse listener is defined in the Applet API documentation. The language-oriented perspective is based on the premise that the application developers think about these concepts when programming, even though they might not be explicitly represented by the available programming language constructs.

Framework-specific modeling languages (FSMLs) are explicit representations of the domain-specific concepts provided by framework APIs [9]. In FSMLs concepts are formalized as *feature models* [61], [30, ch. 4]. A feature model is a hierarchy of *features* that are concept properties. The model further specifies whether each feature is variable or not. For example, the property of being a mouse listener is optional for an Applet, whereas subclassing `Applet` is essential for every Applet. A concept instance is characterized by a *feature configuration*, which is the particular set of features that the instance possesses. A concept instance is *legal* with respect to the concept definition given as a feature model if and only if the feature configuration of the instance satisfies all the constraints imposed by the feature model. In essence, feature hierarchies and constraints define the structure of the concepts, i.e., the *abstract* syntax of an FSML. The semantics of features is defined by *mapping definitions* that relate the features to structural and behavioural patterns that implement the features in the completion code. The mapping definitions are attached to individual features and they state that, for example, a feature corresponds to a call to a particular method or a particular method calling order. Mapping definitions use *mapping types*, which are generic kinds or feature-to-code-pattern cor-

respondences, such as, correspondence to classes or method calls. Mapping types may define *mapping parameters* that must be set in mapping definitions in order to define an actual correspondence. Consequently, we refer to the feature model together with mapping definitions attached to its features as the *metamodel* of an FSML.

This thesis presents a *method for engineering FSMLs* that was extracted post-mortem from the experience of building four such languages. Similar to the Unified Process of object-oriented software development [58], the presented FSML engineering method is *iterative* and *use-case-driven*. FSML use cases play a central role in the method since they embody the value proposition of FSMLs. Consequently, FSML scoping and design decisions are made with respect to the target use cases for the FSML under development. The method considers four main use cases in which FSMLs provide value to the application developers.

The first use case is *framework API understanding* in which the developers learn about the framework-provided concepts and different ways of implementing them by inspecting the feature models and mapping definitions of an FSML. The features and their mapping definitions precisely and concisely specify API usage patterns extracted from the API documentation and expert knowledge and therefore are relevant in understanding the framework API in the scope of the FSML.

The second use case is *completion code analysis and understanding*, in which the developers use an FSML to automatically *reverse engineer* the completion code. Reverse engineering is enabled by *code queries*, which implement mapping definitions in the reverse direction (code to model). The result of reverse engineering is a *framework-specific model*, which contains feature configurations of concept instances identified in the completion code. A framework-specific model provides an overview of the completion code from the viewpoint of the FSML and traceability from features in the model to the matched code patterns. It allows the developers to understand what concept instances are implemented in the completion code and how and where they are implemented. Furthermore, an automated analysis of the framework-specific model can uncover violations of API rules and constraints.

The third use case is *forward engineering* in which the developers first create a framework-specific model and then they generate code that implements the model. Completion code generation is enabled by *code transformations*, which implement mapping definitions in the forward direction (model to code). In general, code transformations and may add, modify, or remove structural or behavioural patterns in the completion code. For completion code generation, code transformations only have to support pattern addition. The value of the use case lies in automating the creation of parts of the application code. Furthermore, the developers can inspect the generated code and learn the default ways of implementing the concepts.

The fourth use case is *completion code evolution*. We distinguish four different scenarios in which FSMLs can support code evolution. At the most basic level, the developers can simply compare framework-specific models extracted at different points in time of the application's life cycle and see how the code has evolved. A

more advanced support is *round-trip engineering* in which both the model and the code can be independently modified and synchronized on demand using incremental updates. Round-trip engineering is useful if only certain changes can be made or are easier to make in either the model or in the code. The third scenario involves migrating application code to a new version of the API. In this case, the FSML is extended with the new features and the old features that are being replaced are marked as *deprecated*. After reverse engineering, the application developers would see that some features are deprecated and need to be migrated. The migration could be supported by an automated transformation. Finally, FSMLs can support migration between two different, but conceptually similar, frameworks.

We devised the FSML engineering method based on our experience in designing and building four FSMLs: *Eclipse Workbench Part Interactions (WPI) FSML*, *Java Applet FSML*, *Apache Struts FSML*, and *Enterprise Java Beans (EJB) FSML*. Each of these languages has a different purpose and value proposition. WPI was the first FSML we built as a proof of concept. Its implementation of round-trip engineering was first hand crafted and served as a basis for building a generic FSML infrastructure that supports declarative specification of mapping definitions. Struts FSML was also first implemented manually and later migrated to the FSML infrastructure. Applet FSML was the first language built entirely declaratively using the infrastructure. EJB FSML is the newest language and it was also specified declaratively. Furthermore, the four languages represent a set of *exemplars*, i.e., a set of representative examples for FSMLs.

The method divides each iteration of the life cycle of an FSML into four phases: *inception*, *elaboration*, *construction*, and *transition*. Each phase involves several development *activities*, and activities are subdivided into *steps*. The thesis focuses on providing concrete examples for the development steps and the outcomes and challenges in each activity based on the four FSMLs. It also provides strategies for making engineering decisions, such as deciding the language scope or the language structure.

Furthermore, we evaluate the method and the exemplar languages. The evaluation is both empirical and analytical. The empirical evaluation involved measuring the precision and recall of reverse engineering and verifying the correctness or forward and round-trip engineering. The evaluation showed that 1) framework-specific models can be extracted from application code with 100% precision and high recall in reverse engineering, 2) the completion code that correctly uses framework's API can be generated in forward engineering, and 3) the model and the code created in multiple iterations of round-trip engineering are equivalent to the model and code created in reverse and forward engineering, respectively. The analytical evaluation focused on the external validity of the method, i.e., evaluating the degree in which the method can be applied for the construction of new FSMLs for other frameworks.

The intended audience of this thesis are framework developers and modeling language developers. Framework developers can learn about the benefits of taking a language-oriented perspective on framework APIs and making framework APIs

more usable through FSMLs. Modeling language developers can learn about the specific steps and guidelines for engineering modeling languages for frameworks. They can benefit from the experience we gathered during the development of four concrete and working sample languages.

The thesis is organized as follows. Chapter 2 provides the necessary background on feature modeling and gives key ideas behind FSMLs. Chapter 3 describes the method for engineering FSMLs. Reverse engineering, forward engineering, and round-trip engineering are evaluated in Chapters 4, 5, and 6, respectively. The method and the experience of building the four FSMLs are discussed in Chapter 7. Related work is presented in Chapter 8. Chapter 9 summarizes the contributions, limitations, and the future work.

1.1 Contributions

We claim the following research contributions of this thesis.

- The concept of FSMLs. This thesis contains the most complete and up-to-date description of the concept of FSMLs to date. The thesis supersedes all previous publications on the topic [6, 9, 13], including the technical reports [8, 10].
- FSML engineering method. This thesis presents language engineering method extracted from the experience of building four exemplar FSMLs.
- Three of the exemplar languages: Applet FSML, Struts FSML, and WPI FSML. EJB FSML was built by Matthew Stephan and it was used only in the preparation of the method.
- Empirical evaluation of the precision and recall of code queries used for reverse engineering. The thesis contains the evaluation of the code queries which are improved versions of the code queries presented in an earlier paper [13].
- An approach to fully-incremental, bidirectional, and homogeneous synchronization called agile round-trip engineering. In this approach, the comparison and reconciliation are performed at the model side.
- Experimental evaluation of the correctness of forward and round-trip engineering. The thesis describes tests used to verify the correctness of code transformations for Java and algorithms used in forward and round-trip engineering.
- The generic FSML infrastructure for building FSMLs and a set of algorithms for reverse, forward, and round-trip engineering.
- A reusable set of mapping types together with code queries and code transformations implementing them.

Chapter 2

Framework-Specific Modeling Languages

2.1 Overview

Using object-oriented frameworks as bases for building applications is hard [23, 57, 67]. Frameworks are designed to be highly reusable and therefore they must be applicable in many different situations, in which they must satisfy particular requirements of the given application. This increases the complexity of the design of the framework. One of the premises of using frameworks for building applications is that application developers do not have to understand the implementation of the entire framework; instead, they should only learn the application programming interface (API) of the framework they are using. The API of a framework provides a set of abstractions, in this thesis referred to as *framework-provided concepts*, and means of implementing them in the application code. After choosing an API concept, application developers instantiate it by writing *framework completion code* according to the rules and constraints specified in the API. However, understanding framework's API as well as writing, understanding, verifying, and evolving the completion code are challenging.

In engineering, *modeling* complex systems is a well established approach to dealing with complexity. A *model* is an artificial representation of real entities, phenomena, or processes that is useful for answering certain questions about them. In software engineering, well known examples of models include *architectural models*, *class models*, and *business process models*. These models allow answering questions about software systems they represent, such as, what are the components of the system and how they interact, what is the class hierarchy of an object-oriented system, and what business process activities are supported by the system.

In this thesis, we propose using *framework-specific models* to support engineering of framework-based applications. A framework-specific model is a representation of a framework-based application that is useful for answering questions related to

the usage of the framework's API by that application. Such questions include: how *is* the application using the framework, is the application using the framework *correctly*, how *should* the application use the framework, and *is* the application using the framework the way it *should* be using it.

Every model needs to be expressed using a formal or an informal *modeling language*. In computer science, a formal language is a set of words over an alphabet. Analogously, a modeling language is a set of models that can be expressed using the constructs of that language. The above definitions are *extensional*, that is, they specify a language as a set of artifacts (words, sentences, models, expressions). In practice, however, languages are defined *intensionally*, that is, by specifying the properties that the artifacts must satisfy in order to belong to the language. Traditionally, textual languages are defined using *grammars*, e.g., context-free grammars, whereas modeling languages are defined using *metamodels*, e.g., class models.

In this thesis, we propose *framework-specific modeling languages* (FSMLs) as languages for expressing framework-specific models. An FSML is a modeling language that formalizes API concepts and constraints provided by the framework the FSML is built for. Also, we propose using *cardinality-based feature models* as abstract syntax definition formalism and, consequently, *feature configurations* as a representation of the framework-specific models. We choose to define the structure of framework-provided concepts using feature modeling because the notation is well suited for expressing variability, which is an important aspect of FSMLs. Feature modeling has been conceived as a commonality and variability modeling technique in the context of domain analysis and software product lines [61], [30, ch. 4]. The connection of feature modeling to domain analysis is important since a substantial part of FSML engineering can be understood as domain analysis.

Table 2.1 presents four example languages, artifacts that they can express, and syntax definition formalisms used for defining them.

Table 2.1: Examples of languages, their artifacts, and language definition formalisms

Language	**Artifact**	**Syntax definition formalism**
a*	string of letters `a`	regular expression
Java	program	BNF context-free grammar
UML	model, e.g., class, activity	MOF class model
Applet FSML	framework-specific model (feature configuration)	cardinality-based feature model

Figure 2.1 presents an overview of model-supported engineering of framework-based applications using FSMLs. Boxes represent entities, black arrows represent relationships, callouts represent contents of entities, and yellow (thick) arrows represent use cases. The figure presents key relationships between applications, frameworks, and framework-specific models and languages. An application *uses* a framework, that is, it implements instances of concepts provided by the framework's API. A framework-specific model *represents* the instances of the concepts

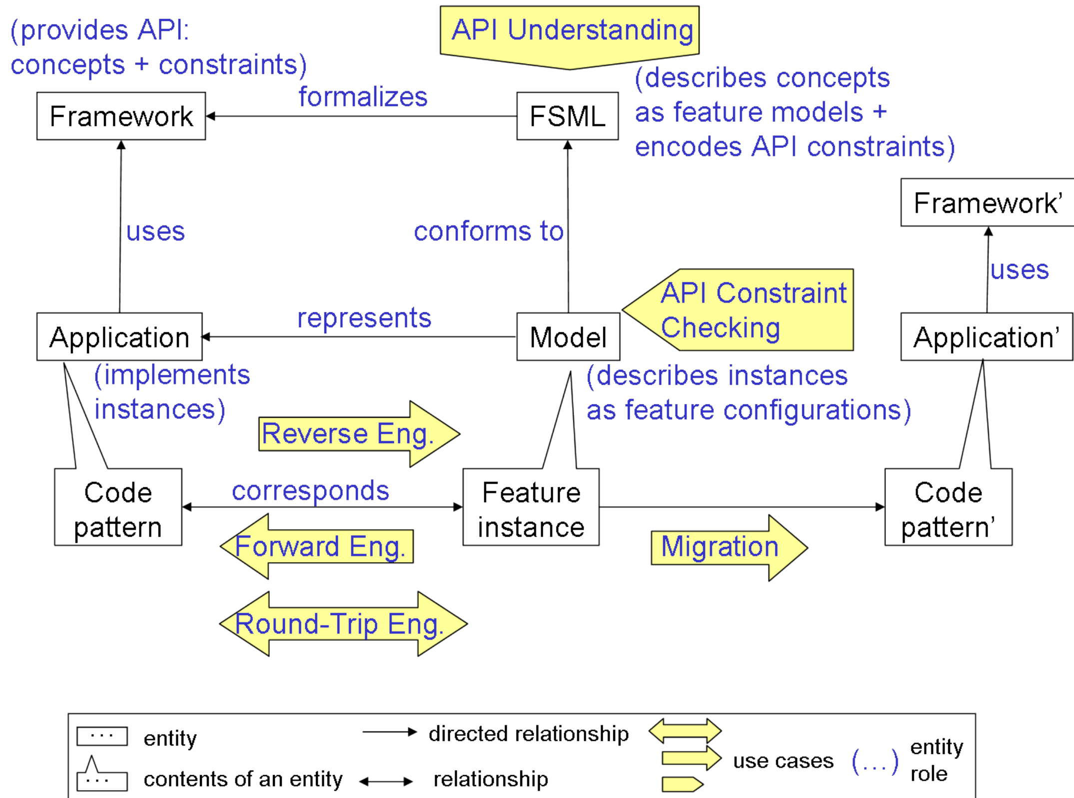

Figure 2.1: Overview of model-supported engineering of framework-based applications using FSMLs

implemented in the application code as feature configurations. An FSML *formalizes* API concepts and constraints as feature models. A framework-specific model should *conform to* an FSML, that is, feature configuration should satisfy all constraints specified in a feature model. Finally, a framework-specific model contains feature instances and the application code consists of code patterns that implement different functionalities of the application. Code patterns can be structural (e.g., a field) and behavioural (e.g., a method call in the control flow of an object, order of method calls).

A framework-specific model *represents* an application because feature instances from a feature configuration correspond to code patterns that implement them in the application code. In FSMLs, the correspondence relationship between the features and code patterns is specified using *mapping definitions* as shown on Figure 2.2. The figure is divided into four parts: *application*, *model*, *FSML*, and *mapping interpreter*, each representing a different entity of the approach. In the figure, a feature instance *represents* a code pattern and a code pattern *implements* a feature instance. This correspondence is specified using mapping definitions that precisely specify what code patterns implement instances of a given feature using a *pattern expression*. Mapping definitions may also specify values of some additional parameters that may be required by code queries of transformations. Each mapping

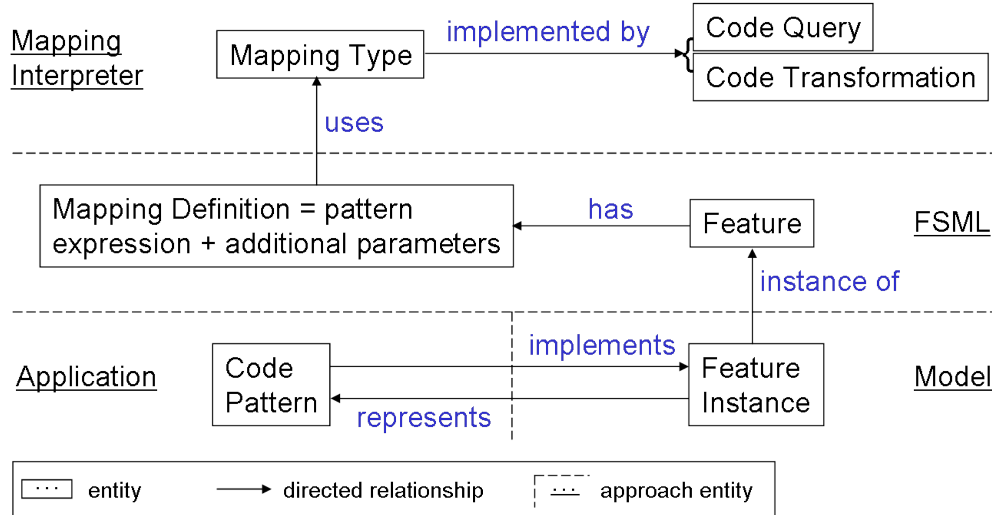

Figure 2.2: Mapping definitions specify the feature-to-code-pattern correspondence

definition *uses* a particular *mapping type*. Mapping types are generic and reusable and they represent different kinds of feature-to-code-pattern correspondences, such as, a correspondence to Java classes, method calls, or XML elements. Mapping types define *parameters* that are set in mapping definitions to define a particular correspondence, such as, a parameter *method signature* used to specify a correspondence to method calls with the given signature. The difference between mapping types and definitions is similar to the difference between SQL language constructs (e.g., select <p> from <t>, where p and t are parameters) and particular SQL queries (e.g., select * from 'transactions').

We refer to the feature model together with mapping definitions as the *metamodel* of an FSML. The metamodel can be used in each of the four main use cases introduced in Section 1: *framework API understanding*, *completion code analysis and understanding*, *forward engineering*, and *completion code evolution*. In Figure 2.1, we present framework API understanding and forward engineering use cases directly. *Reverse engineering* and *API constraint checking* belong to the completion code analysis and understanding use case. *Round-trip engineering* and *migration* belong to the completion code evolution use case. In framework API understanding, the metamodel is inspected manually. In reverse, forward, round-trip engineering use cases the metamodel is interpreted by algorithms for reverse, forward, and round-trip engineering, respectively, implemented in the generic FSML infrastructure. Migration is performed using specialized forward engineering that creates new code patterns in a new application that uses a new version of the frameworks or even an entirely new framework.

Since the mapping types are only specifications, they are *implemented by* code queries for reverse engineering and code transformations for forward and round-trip engineering (cf. Figure 2.2). Mapping definitions provide values of parameters defined by mapping types and required by code queries and transformations. The top

part of Figure 2.2, labelled as *mapping interpreter*, illustrates that mapping types are generic, reusable, implemented in a mapping interpreter, and that they are not part of the metamodel of any FSML. We refer to the time when the metamodel is interpreted by the algorithms as *FSML execution*. During FSML execution, the generic FSML infrastructure delegates the execution of code queries and transformations to pluggable mapping interpreters, which interpret mapping definitions attached to the features.

In the following sections we delve into the details of feature models and configurations, mapping types and definitions, code queries and transformations, and metamodel interpretation algorithms. We also briefly present the generic FSML infrastructure that is used for implementing FSMLs.

2.2 Feature Models of Abstract Syntax

In this chapter we use `Java Applet` as an example concept from the Java Applet framework and we show how this concept can be formalized as an FSML. Figure 2.3 shows fragments of the Java Applet Tutorial [90] that we use.

> "An applet is a special kind of Java program that a browser enabled with Java technology can download from the internet and run."
>
> "An applet must be a subclass of the `java.applet.Applet` class, which provides the standard interface between the applet and the browser environment."
>
> "Swing provides a special subclass of Applet, called `javax.swing.JApplet`, which should be used for all applets that use Swing components to construct their GUIs."
>
> "Life Cycle of an Applet: Basically, there are four methods in the Applet class on which any applet is built.
>
> `init`: This method is intended for whatever initialization is needed for your applet. It is called after the param attributes of the applet tag.
>
> `start`: This method is automatically called after init method. It is also called whenever user returns to the page containing the applet after visiting other pages.
>
> `stop`: This method is automatically called whenever the user moves away from the page containing applets. You can use this method to stop an animation.
>
> `destroy`: This method is only called when the browser shuts down normally."
>
> "To draw the applet's representation within a browser page, you use the `paint` method."
>
> "Parameters are to applets what command-line arguments are to applications. They allow the user to customize the applet's operation.
>
> Applets get the user-defined values of parameters by calling the Applet `getParameter` method. "
>
> "You should also implement the `getParameterInfo` method so that it returns information about your applet's parameters."

Figure 2.3: Fragments of the Java Applet Tutorial [90] used as a running example in this chapter

In FSMLs, we use *feature models* to model framework API concepts by decomposing them into their characteristic *features* so that a particular instance of the concept can be described by a particular selection of features it possesses. In our example, features will represent the quoted fragments of the documentation, for example, whether an applet extends the `JApplet` class or whether it uses named parameters. A particular instance of the concept, that is a particular applet, will have certain features and may be missing other ones, for example, an applet may implement the method `init`, use two parameters named `'WIDTH'` and `'HEIGHT'`, and not implement the method `stop`.

A feature model is 1) a hierarchy of features and 2) a list of additional constraints that cannot be expressed using the hierarchy. The root of the hierarchy is referred to as a *concept* or a *root feature*. A feature model encodes a set of all possible legal *feature configurations*, which contain *feature instances*. A feature configuration is said to be legal if it satisfies a conjunction of all constraints encoded in the feature hierarchy and the additional constraints. The relation between a feature model and a feature configuration is similar to the relation between a class model and an object model, in which objects are instances of classes and the object model is legal if it satisfies all constraints encoded in the class model. Also, the feature hierarchy is semantically similar to the containment hierarchy in a class model in that an instance of a feature cannot exist in a feature configuration without an instance of its parent feature (except the root feature) the same way as a part object cannot exist without its container object.

Figure 2.4 presents a fragment of the feature model of the concept `Applet`. The left column contains a hierarchy of features represented using indentation: subfeatures are further to the right. The right column contains explanations of elements in the left column.

We use a slightly extended form of the *cardinality-based feature modeling* [31, 33]. In this variant of feature modeling, each feature has a *feature cardinality*. The cardinality is an interval specifying how many instances of a given feature may be present as children of an instance of its parent feature in a feature configuration. *Optional* features have the cardinality of [0..1]. *Mandatory* features have the cardinality [1..1]. *Prohibited* features are denoted by [0..0]. *Multiple* features have the upper bound of the interval greater than 1 or no upper bound as indicated by the Kleene star (*). For example, the cardinality [2..*] specifies that at least two clones (instances) of the feature should be present in the feature configuration as children of every instance of the parent feature. In Figure 2.4, `extendsApplet` is a mandatory feature, `extendsJApplet` is optional, and `parameter` is an optional multiple feature.

Features can be also organized into *feature groups*. Each feature group has a *group cardinality*, which is an interval specifying how many of the grouped features can be present in a feature configuration. *Exclusive-or* (XOR) feature groups have the cardinality of <1-1>, and *inclusive-or* (OR) feature groups have the cardinality of <1-k>, where k is the number of features in the group. A feature group with

Feature hierarchy	*Explanation*
`Applet`	*concept (root feature)*
`  [1..1] name (String)`	*mandatory feature with attribute*
` ![1..1] extendsApplet`	*essential feature*
`    [0..1] extendsJApplet`	*optional feature*
`  [1..1] lifecycleMethods`	*mandatory feature*
`   !<1-5>`	*essential* OR *feature group*
`      [0..1] init`	
`      [0..1] start`	
`      [0..1] paint`	*optional grouped features*
`      [0..1] stop`	
`      [0..1] destroy`	
`  [0..*] parameter`	*optional multiple feature*
`    [0..*] name (String)`	*multiple feature with attribute*
`  [0..1] providesParamInfo`	*optional feature*
`  [1..1] infoForParams`	*mandatory feature*

Figure 2.4: Feature model for the concept applet

the cardinality <0-*k*> is equivalent to *k* optional subfeatures. The sample model in Figure 2.4 contains an OR-group with five features, that is, at least one feature from that group has to be present in the configuration.

Features may also have a *type*, which means that a single value of that type can be associated with a given feature instance in a feature configuration. A feature with a type is essentially an *attribute* whose value is set in configuration. For example, the sample model in Figure 2.4 contains two features having the type `String`. Some attributes may also point to other features in the configuration; we refer to such attributes as *reference attributes*.

Constraints that cannot be encoded in the hierarchy can be specified separately as *additional constraints* or they can be included in the hierarchy as mandatory features. For example, a constraint may specify that an instance of an optional feature in one part of the hierarchy may require an instance of another optional feature in a different part of the hierarchy. More complex constraints may involve Boolean expressions and constraints over sets of feature instances [33]. In Figure 2.4, the mandatory feature `infoForParams` corresponds to the constraint `../parameter→../providesParamInfo`, that is, whenever at least one instance of the feature `parameter` is present in a feature configuration, an instance of the feature `providesParamInfo` should also be present. Such a constraint could also be specified separately as ∀`a:Applet . a/parameter→a/providesParamInfo`, that is, the constraint must hold for every instance of the concept `Applet`. Note that both constraints use *path* expressions for feature hierarchy navigation, which are similar to *XPath* or Unix file system paths.

The notation we use also supports *feature inheritance*, which is similar to class inheritance and allows a feature to inherit the subfeatures of another feature [20].

We denote feature inheritance by `-|>`, e.g., `ViewPart -|> Part` means that the feature `ViewPart` inherits subfeatures of the feature `Part`. Features can also be abstract in the same sense as classes, that is, they cannot be instantiated. For example, the feature `Part` is abstract. Also, the Liskov substitution principle holds, that is, inheriting features can be used in place of the feature they inherit from. For example, a feature with a reference attribute `[1..1] provider (Part)` may point at exactly one instance of any feature that inherits from feature `Part` in a feature configuration but not at an instance of feature `Part` since it is abstract.

For the purpose of FSML modeling, we also introduce the notion of *essential features* and *essential feature groups*. Essential features and feature groups have the lower bound of the cardinality greater than zero and that lower bound cannot be violated at any time. The notion of essential constraint is similar to the philosophical notion of *essential property*, that is, a property that the instance of a concept has always and in every possible world; otherwise it is not an instance of that concept [95]. We mark the essential features and feature groups using the exclamation point (!). Note that the essentiality constraint, similarly to the cardinality constraint, is specified with respect to the direct parent feature only, that is, it constraints the presence of instances of the parent feature.

For example, the feature `extendsApplet` is an essential feature of `Applet` and the group `<1-5>` is an essential feature group of the feature `lifecycleMethods`. Note that without an instance of `extendsApplet` feature, an instance of the concept `Applet` is not an applet. Similarly, without at least one instance of any of the life cycle methods features (`init`, `start`, ...) an instance of the feature `lifecycleMethods` should never be present in a configuration.

Semantically, a feature model describes a set of all legal feature configurations, that is, configurations that satisfy all constraints encoded in the feature model (hierarchy, cardinality, and additional constraints). However, for practical reasons, we must be able to express illegal feature configurations in which some constraints can be violated, which allows us to check for these violations. By introducing the notion of essential features and feature groups whose cardinality constraints cannot be violated at any time and allowing non-essential constraints to be violated, we define a superset of the set of legal feature configurations. Such a superset allows for expressing incorrect concept instances while making sure that the essential constraints are never violated.

Figure 2.5 presents two sample feature configurations of the feature model from Figure 2.4. The first configuration is *legal* because it satisfies all constraints imposed by the feature model. The second configuration is *illegal* because the cardinality constraint of the feature `lifecycleMethods` is violated: a mandatory feature is missing. The feature `lifecycleMethods` cannot be present because the group cardinality of the essential group is violated. Note that if the essential feature `extendsApplet` had been missing, the concept instance `Applet` would not have existed.

Feature modeling is closely related to class modeling, which is currently more

Feature Instance Hierarchy	*Explanation*
[1] Applet	*concept instance*
[1] name ('MyApplet')	*attribute value*
[1] extendsApplet	*essential feature present*
[0] extendsJApplet	*optional feature missing*
[1] lifecycleMethods	*mandatory feature present*
!<1-5>	*group constraint satisfied*
[1] init	*grouped feature present*
[1] start	*grouped feature present*
[0] paint	
[0] stop	*grouped features missing*
[0] destroy	
[1] parameter	*clone of a multiple feature*
[1] name ('color')	*attribute value*
[1] parameter	*clone of a multiple feature*
[1] name ('width')	*clone with an attribute value*
[1] name ('height')	*clone with an attribute value*
[1] providesParamInfo	*optional feature present*
[1] infoForParams	*mandatory feature present*
[1] Applet	*concept instance*
[1] name ('MyApplet2')	*attribute value*
[1] extendsApplet	*essential feature present*
[1] extendsJApplet	*optional feature present*
[0] lifecycleMethods	*mandatory feature missing*
!<1-5>	*group constraint violated*
[0] init	
[0] start	
[0] paint	*grouped features missing*
[0] stop	
[0] destroy	
[0] parameter	*no clones*
[0] name (String)	*no values*
[1] providesParamInfo	*optional feature present*
[1] infoForParams	*mandatory feature present*

Figure 2.5: Feature configurations of two applet instances

commonly used to define the abstract syntax of modeling languages. Cardinality-based feature modeling with feature reference attributes and feature inheritance nearly coincides with class modeling [32, 66]. An important property of feature modeling that distinguishes it from class modeling is improved support for variability modeling due to feature groups. Furthermore, feature models are rendered hierarchically with a focus on the concept decomposition into features, whereas the graph-like rendering of class models is more geared towards showing relations among classes. Feature models are also closely related to grammars and logic [22, 31, 34].

This relation has enabled the application of grammar and logic-based algorithms in the analysis of feature models and in feature configuration.

2.3 Mapping Definitions

Although feature models adequately represent the structure and variability of the concepts, the feature model notation treats features merely as symbols devoid of further semantics. In FSMLs, we specify the semantics of features using *mapping definitions* that define the correspondence between features and structural and behavioural code patterns in the completion code [13].

Traditionally, semantics of languages is specified by giving a *semantic domain* and a *semantic mapping* [52]. Semantic mapping maps syntactic elements to the elements of the semantic domain. For example, assuming a semantic domain of natural numbers, a semantic mapping would map expressions in some arithmetic language (e.g., `12 add: 4`) into their interpretation in the semantic domain (e.g., 16, that is the result of 12 + 4).

In the case of FSMLs, the semantic domain is structural and behavioural code patterns and the semantic mapping maps features to code patterns. Mapping definitions, however, are only a part of the semantic mapping because they only map a single feature to a single code pattern. Semantic mapping, on the other hand, recursively maps a single feature to the code patterns that that feature and all of its subfeatures correspond to. Formally defining the semantic domain and the semantic mapping, however, remains future work. Here we only provide intuitions for such a formalization.

Figure 2.6 presents mapping definitions for the features from Figure 2.4. The mapping definitions are presented in angle brackets using a Smalltalk-like notation. They use predefined *mapping types* that represent basic kinds of feature-to-code-pattern correspondences. Mapping types used in this example are presented in Tables 2.2 and 2.3. For example, the mapping definition <`class`> for the concept `Applet` specifies that the instances of the concept correspond to Java classes.

Table 2.2: Selected mapping types for structural code patterns from Table 4.1

Structural Pattern Expression	**Structural Element(s) Matched**
class	matches a Java class
c fullyQualifiedName	matches the fully qualified name of the class *c*
c assignableTo: *t* [concrete: r] [local: *p*]	matches if objects of the class *c* are assignable to the type *t*. The optional parameter *r* specifies that only concrete classes should be matched. The optional parameter *p* specifies that only source types of the project *p* should be matched
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

Each mapping type has a number of parameters that must be set to define an actual correspondence to code patterns. In mapping definitions, the parameter

Feature hierarchy	***Feature correspondence explanation***
Applet <class>	*concept instance corresponds to a Java class*
[1..1] name (String) <fullyQualifiedName>	*value of the feature is a fully qualified name of the context class*
![1..1] extendsApplet <assignableTo:'Applet'>	*feature is present if the context class is a subtype of Applet*
[0..1] extendsJApplet <assignableTo:'JApplet'>	
[1..1] lifecycleMethods	*a feature used for grouping, no mapping definition*
!<1-5>	
[0..1] init <methods:'void init()'>	*each grouped feature corresponds to a non-inherited method of the context class*
[0..1] start <methods:'void start()'>	
[0..1] paint <methods:'void paint(Graphics)'>	
[0..1] stop <methods:'void stop()'>	
[0..1] destroy <methods:'void destroy()'>	
[0..*] parameter <callsReceived:'String getParameter(String)' location:'void init()'>	*clones correspond to calls received by objects of the context class*
[0..*] name (String) <valueOfArg:1>	*value of the feature is the value of the first argument of the context method call*
[0..1] providesParamInfo <methods:'String[][] getParameterInfo()'>	
[1..1] infoForParams <constraint:../parameter requires:../providesParamInfo>	*constraint*

Figure 2.6: Mapping definitions for features from the applet feature model

values can be provided statically or retrieved from other features during FSML execution. The first option involves specifying the parameter values directly in the mapping definitions. For example, the type name Applet is specified explicitly as a parameter value in <assignableTo: 'Applet'>. The second option involves specifying a feature as a parameter, in which case the code pattern corresponding to that feature during FSML execution will be used as the parameter value. The feature to be used for the retrieval of the parameter value is specified by an absolute or relative path expression, such as, ../parameter in Figure 2.6.

In addition to specifying parameters explicitly, parameter values can also be determined implicitly using a *context mechanism*. The context mechanism retrieves the value for a parameter from the instance of the closest parent feature with the required mapping type. For example, the mapping type *c* fullyQualifiedName requires a class as the value of its parameter *c* (cf. Table 2.2). However, the mapping definition for Applet's subfeature name is <fullyQualifiedName>; that is, it does not explicitly specify the feature corresponding to a class. Therefore, the context mechanism selects Applet as the feature since it is the closest parent with a mapping type that matches a class, which means that the class corresponding to an instance of Applet is used as the parameter value to fullyQualifiedName. That

Table 2.3: Selected mapping types for behavioural code patterns from Table 4.2

Behavioural Pattern Expression	Run-time Event Pattern(s) Matched
c callsTo: s receiver: r [statement: s]	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. The optional parameter s specifies whether only method calls which are individual statements should be matched.
c callsReceived: s	matches method calls to methods with the signature s received by objects assignable to the class c
mc valueOfArg: i	matches run-time values of the ith argument of the method call mc

class is also referred to as the *context class*. As a result, the mapping definition <`fullyQualifiedName`> specifies that the feature `name` corresponds to the fully qualified name of the class corresponding to the instance of `Applet`. By similar reasoning, the context class is also implicitly the parameter value in all the remaining mapping definitions in Table 2.6 except the subfeature `name` of `parameter`. The method call required as a value for the parameter `mc` for the mapping definition <`valueOfArg: 1`>, is the method call that corresponds to the instance of its parent feature. Note that the context mechanism is redundant because an explicit path to the appropriate parent feature can always be given. However, the mechanism greatly simplifies the metamodels and it reflects the fact that, in feature modeling, subfeatures are properties of their parent features.

An instance of a mandatory, optional, or essential feature in a configuration directly corresponds to at least one pattern in the code. Each instance (clone) of a multiple feature, e.g., `parameter`, directly corresponds to exactly one pattern in the code. Note that the semantic mapping specifies the actual correspondence, that is, every feature instance directly corresponds to a code pattern specified by its mapping definition and indirectly corresponds to all code patterns of its subfeatures.

For example, underlined code patterns in Figure 2.7 implement feature instances from the first configuration in Figure 2.5. For clarity, we only underlined names of class and method declarations instead of the whole declarations. In summary, the concept instance `Applet` corresponds to the entire class declaration (lines 1-14), the instance of `name` corresponds to the class' name (line 1), the instance of `extendsApplet` corresponds to the super class declaration (line 1). Instances of `init`, `start`, and `providesParamInfo` correspond to method declarations on lines 2-9, 10, and 11-13, respectively. Two instances of `parameter` correspond to method calls on lines 3 and 8 which are received by the applet. The instance of `name` for the first parameter corresponds to the first argument of the method call on line 3. The two instances of `name` for the second parameter correspond to the two possible values (lines 4 and 6) of the first argument of the method call on line 8.

2.4 Code Queries and Transformations

Mapping types define only the feature-to-code-pattern correspondence and they are not directly executable. In our implementation, we realized mapping types by

```
1  public class MyApplet extends Applet {
2    public void init() {
3      String color = getParameter("color");
4      String paramName = "width";
5      if (...)
6        paramName = "height";
7      ...
8      String dimension = getParameter(paramName);
9    }
10   public void start() { ... }
11   public String[][] getParameterInfo() {
12     return ...;
13   }
14 }
```

Figure 2.7: Sample code described by the first configuration from Table 2.5

implementing code queries and code transformations for each type. Code queries match structural and behavioural patterns in the completion code and code transformations incrementally add, modify, and remove code patterns. Code transformations may require additional information which is usually abstracted away by code queries. Such information is specified using *forward parameters* of mapping types.

For example, a code query for the `callsReceived` mapping type from Table 2.3 matches method calls to methods with a given signature received by objects assignable to a given type (cf. Table 2.5), whereas a code transformation for that mapping type adds such a method call in the context class (cf. Table 2.4). The signature of a target method in which the method call is to be added is specified using the forward parameter `location`, e.g., `location: 'void init()'` (cf. Figure 2.6).

Mapping types are defined for a certain artifact type, such as Java or XML, and so are code queries and transformations. That way, the FSML approach is not tied to a particular artifact type and providing support for a new artifact type amounts to defining new mapping types and implementing code queries and transformations. The generic FSML infrastructure is artifact-type-agnostic, i.e., it supports pluggable *mapping interpreters* that implement mapping types for a given artifact type. To date, we developed three mapping interpreters: a *Java mapping interpreter* for Java files, a *Plug-in mapping interpreter* for plugin.xml files of Eclipse plug-ins, and an *XML mapping interpreter* for XML documents.

It is important to note that, whereas mapping definitions specify an exact correspondence between the features and code patterns, code queries and code transformations may only approximate that correspondence. For example, a code query for the mapping type `callsTo` (cf. Table 2.3) locates method calls in the control

flow of objects of a given class and, by the very nature of static analysis, the control flow graph used during search is an approximation of the run-time control flow [13].

Mapping types used in the exemplar languages and code queries implementing them are presented in Chapter 4. Code transformations implementing the mapping types are presented in Chapter 5.

2.5 Metamodel Interpretation Algorithms

The algorithms implemented in the generic FSML infrastructure were designed to leverage the structure of feature models which are trees and where subfeatures are properties of parent features. In a feature configuration, a feature cannot exist without its parent feature and a parent feature cannot exist without all of its essential subfeatures. Therefore, the algorithms process the metamodel in the depth-first manner. Additionally, the subfeatures of a feature are processed in the order they appear in the metamodel. Although, using the order of features to control the order of feature processing is somewhat arbitrary because the order could be automatically determined by analyzing dependencies between mapping definitions, we did not find it too restrictive and it simplified the design of the algorithms.

The processing of a feature involves two phases: parameter evaluation and execution of code queries and transformations. For a given feature, its mapping definition is processed by an appropriate mapping interpreter, that is, an interpreter that implements the mapping type used in the mapping definition (cf. Figure 2.2). Parameter evaluation consists of computing default values for optional static parameters and retrieving values from other features for dynamic parameters; the latter involves either using context mechanism or navigating to the target feature using the explicit path as described in Section 2.3. Note that the value can be retrieved only if the target feature has already been processed before processing the given feature: this is always the case for context mechanism, because the parent feature is always processed first. When using explicit paths for specifying target features the FSML designer must ensure, by controlling the order of features, that the target feature will be processed first.

The algorithms assume that the metamodel is both syntactically correct and well typed. Well-typedness is related to the requirements imposed by the mapping types used in mapping definitions: (1) all required static parameters must be set (optional static parameters can be left unspecified); (2) target features specified by path expressions for dynamic parameters must exist, they must have mapping definitions using the required mapping type, and they must be processed before the feature with the given mapping definition; and (3) if the context mechanism is used for a dynamic parameter (i.e., the path expression is not specified), then the parent feature with a mapping definition using the required mapping type must exist. Design-time type checking is currently not implemented; instead, run-time exceptions are thrown if any of the conditions mentioned above is violated. Additionally, certain checks cannot be executed at design time. For example, verifying

the existence of classes and interfaces referred to by fully qualified name in mapping definitions can be performed only in the context of a particular project at run-time.

The algorithms also actively enforce the satisfaction of the essentiality constraint by removing an instance of the parent feature whenever the lower bound of an essential subfeature or subfeature group is not satisfied. The entire subtree must be removed because a feature instance cannot exist without an instance of its parent feature. Note that every essential feature also is, by definition, mandatory. However, the cardinality constraint of mandatory features is not actively enforced, that is, missing instances of mandatory features cause errors in the configuration of instances of their parent features instead of causing the removal of the parent instance. This active enforcement of the essentiality constraint is similar to the enforcement of the *containment* constraint in class models, whereby a part object cannot exist without its container object, and therefore all part objects are removed together with their container object.

2.5.1 Forward engineering

Forward engineering creates code for a given framework-specific model and is driven by the model. The algorithm simply traverses the feature configuration in a depth-first manner starting from the instance of the root feature and executes code transformations for every feature with a mapping definition.

Table 2.4 presents selected code transformations that implement mapping types used in the example from Figure 2.6. Because forward engineering is executed for an existing model, some code transformations retrieve values of their parameters from other features. This allows specifying, for example, class or method names as attribute values of feature instances. For example, the code transformation `addClass` retrieves the fully qualified name of the class to be created from subfeatures with mapping types `className` and `qualifier` or from a subfeature with mapping type `fullyQualifiedName`.

For example, for the first configuration from Figure 2.5, the algorithm first creates a Java class called `MyApplet` using `addClass`, next it adds the `extends Applet` superclass declaration and the necessary import statement `import java.applet.Applet;` using `addAssignableTo`, it creates two methods `void init()` and `void start()` using `addMethod`, it creates two method calls `getParameter(null)` using `addCallTo`, it replaces the first argument of the first method call with `"color"` and of the second call with `name0`, where `name0` is a local variable assigned twice with values `"width"` and `"height"` using `addArgVal`. Both method calls are inserted into the method `init` as specified using a forward parameter `location` (cf. Table 2.6). Finally, the algorithm creates a method `String[][] getParameterInfo()` using `addMethod`. Figure 2.8 presents the resulting code. We evaluate forward engineering in Chapter 5.

Code Transformation Expr. | Defaults	**Result**
p addClass: n [in: q] | q=”	creates a compilation unit with a class declaration named n in package q. Retrieves values of the parameters n and q from subfeatures with mapping types `className` and `qualifier` or `fullyQualifiedName`
c addAssignableTo: t [concrete: e] | e=true	If t is an interface, adds a `c implements t` superinterface declaration or adds t to the existing list of implemented interfaces. If t is a class, adds a `c extends t` superclass declaration. If e=true, adds implementations of the unimplemented methods of the superinterface or an abstract superclass
c addMethod: s [name: n]	adds a method declaration of signature s in the class c. If method name n is given, replaces the name from the signature s with n. If the signature contains * for the method name, the parameter n is mandatory. Retrieves the value of the parameter n from a feature with the mapping type `methods`
c addCallTo: s [receiverExpr: r] location: l [position: p] | r=”, p=after	creates a method call to a method with the signature s with the receiver expression r in the method of signature l of the class c at the position $p \in$ {before, after}
mc addArgVal: i [values: v]	adds values of the ith argument of the method call mc. Adds a literal for a single value. For multiple values create a variable and multiple assignments with values v

Table 2.4: Selected code transformations for the `class`, `assignableTo`, `methods`, `callsReceived`, and `valueOfArg` mapping types

```
import java.applet.Applet;
public class MyApplet extends Applet {
  void init() {
    getParameter("color");
    String name0 = "width";
    name0 = "height";
    getParameter(name0);
  }
  void start() {
  }
  public String[][] getParameterInfo() {
    return null;
  }
}
```

Figure 2.8: Code generated for the first configuration from Figure 2.5

2.5.2 Reverse engineering

Reverse engineering creates a framework-specific model for the given application code. Reverse engineering is driven by the metamodel of an FSML: the algorithm traverses the metamodel, executes code queries for features in the metamodel, and creates instances of these features in the model for the code patterns matched by the queries. After creating an instance of a feature in the model, a traceability link is established between the feature's instance and the corresponding code pattern. Traceability links enable bidirectional navigation between the model and the code. The algorithm begins with creating instances of the root feature. In general, there are five ways in which an instance of a feature is created.

(1) If a feature does not have any mapping definitions attached, an instance of the feature is created and its subfeatures are processed next. For example, an instance of the feature `lifecycleMethods` is created this way.

(2) If a feature has a mapping definition that uses a mapping type for which a code query can be directly executed, an instance is created for each result of the query if a feature is multiple or a single instance is created if the result is non-empty otherwise. For example, instances of the feature `parameter` are created for every method call matched by the query for the `callsRec` mapping type.

(3) If a feature has a mapping definition that uses a mapping type for which a code query cannot be directly executed, the query for an essential feature is executed in the *essential mode*. In the essential mode, instances of the parent feature are created based on the results of the code query for a subfeature. For example, instances of the concept `Applet` are created based on the results of the query for the feature `extendsApplet`, that is, an instance of `Applet` is created for every Java class that extends the `java.applet.Applet` class.

(4) If a feature has a mapping definition that uses a mapping type for which a code query cannot be directly executed and the feature has a *base concept reference* subfeature, an instance of the feature is created for every instance of the feature referenced through the base concept reference and the reference attribute is set to point at that feature. The base concept reference mechanism is used for modularizing the metamodels of FSMLs and it allows for composition of multiple languages. For example, the feature `provider` (Figure 3.3, line 63) is a base concept reference. An instance of the concept `AdapterProvider` will be created for every instance of the concept `Part` and the reference attribute of the feature `provider` will be set to the instance of `Part` for which the adapter provider was created. As a result, an instance of `AdapterProvider` will correspond to the same class as the instance of `Part` corresponds to. Note that, due to ordering of features, all instances of `Part` will be created before instances of `AdapterProvider`.

(5) If a feature has an attribute and no subfeatures, an instance of the feature is created for every value returned by the code query and the feature's attribute is set to that value. For example, an instance of the feature `name` will be created for each value returned by the query for the mapping type `valueOfArg` [1].

(6) Finally, if a feature corresponds to a constraint, an instance of the feature is created if the constraint is satisfied. For example, an instance of the feature `infoForParams` (cf. Figure 2.6) will be created if the `requires` constraint is satisfied.

After creating an instance in one of the ways mentioned above, the instance can be removed if any of its essential features are missing or essential feature groups are violated (we say that an instance is excluded from the configuration). If an instance is not removed, the traceability links are established between the instance and the matched code pattern(s).

[1] Static analysis may return multiple values if the argument is a variable (cf. Section 4))

The last phase of reverse engineering is evaluation of model queries and it takes place after the entire model has been created. Model queries can set the values of feature attributes to the values of other features or set reference attributes to point at other features. For example, the feature `actionImpl` (Figure 3.4, line 19) points at an instance of the concept `ActionImpl`.

The essential mode significantly reduces the time of reverse engineering. For example, the mapping definition <`class`> specifies that instances of the concept correspond to Java classes and executing a query directly would match all classes in the system and an instance of applet would be created for each class. All instances that correspond to classes that do not extend the class `Applet` would then be excluded from the configuration because they would not have the essential feature `extendsApplet`. Using the essential mode avoids unnecessary processing of instances for which essential features are missing. Also note that it is not necessary to execute queries in essential mode for each essential feature and intersect resulting sets of instances because the intersection will be computed by removing all instances that have at least one of the remaining essential features missing. This way only instances that have all their essential features will remain.

Not every code query can be executed in the essential mode. Furthermore, whether a query will be executed in the essential mode depends on the structure of the feature model and whether the feature is essential. For example, the code query for the feature `extendsJApplet` (i.e., `getAssignableTo`) will not be executed in the essential mode since the feature is not essential. An instance of the feature will be created only if the context class extends the class `JApplet` (cf. (2)).

Table 2.5 presents code queries that implement mapping types used in the example from Figure 2.6. Out of these, only the query `getAssignableTo` can be executed in the essential mode.

Code Query Expr.	**Result**
c getAssignableTo: *t* [concrete: *e*] | e=true	true if the class *c* is assignable to the type *t*. In essential mode, set of classes assignable to the type *t*, limited to concrete classes only if *e*=true.
c getFullyQualifiedName	fully qualified name of the class (*c*).
c getMethods: *s*	set of methods of signature *s* in the class *c*. The signature *s* may contain * for the method name.
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
mc getArgValConstantProp: *i*	set of values of the ith argument of the method call *mc* retrieved using interprocedural constant propagation limited in scope to the class that contains the called method

Table 2.5: Selected code queries for the `class`, `assignableTo`, `methods`, `callsReceived`, and `valueOfArg` mapping types

For example, reverse engineering the code from Figure 2.8 produces the first configuration from Figure 2.5 and proceeds as follows. First, one instance of the concept `Applet` is created for the only Java class that extends `java.applet.Applet` by executing `getAssignableTo` for the essential feature `extendsApplet` in the essential mode (cf. (3)). Next, the value of the feature `name` is set to the fully

qualified name of the context class (cf. (5)) using `getFullyQualifiedName`. Next, the feature `extendsApplet` is instantiated and its subfeature `extendsJApplet` is not instantiated using `getAssignableTo`. Next, the feature `lifecycleMethods` is instantiated (cf. (1)) and two of its subfeatures are instantiated (`init` and `start`) using `getMethods`. The feature `paint` is not instantiated because the applet does not implement the method `paint`. The instance of the feature `lifecycleMethods` is not removed because its essential group's cardinality is satisfied. Next, the query `getCallsReceivedTI` for the feature `parameter` matches two method calls. One instance of the feature `parameter` is created for the first method call (cf. (2)) and an instance of the feature `name` is created for and set with the value of the first argument of the context method call using `getArgValConstantProp` (cf. (5)). Analogously, the second instance of the feature `parameter` is created for the second method call (cf. (2)) and two instances the feature `name` are created for each possible argument value (cf. (5)). The feature `providesParamInfo` is instantiated using `getMethods`. Finally, an instance of the feature `infoForParams` is created (cf. (6)). We evaluate reverse engineering in Chapter 4.

2.5.3 Round-trip engineering

Round-trip engineering (RTE) incrementally synchronizes a number of related artifacts. The artifacts are first compared to identify the changes that were made since the last time the artifacts were synchronized. Next, in the reconciliation phase, the identified changes are propagated to the related artifacts in order to reestablish the consistency among the artifacts. We implemented a particular approach to round-trip engineering called *agile round-trip engineering* [9] in which comparison and reconciliation are performed on the model side.

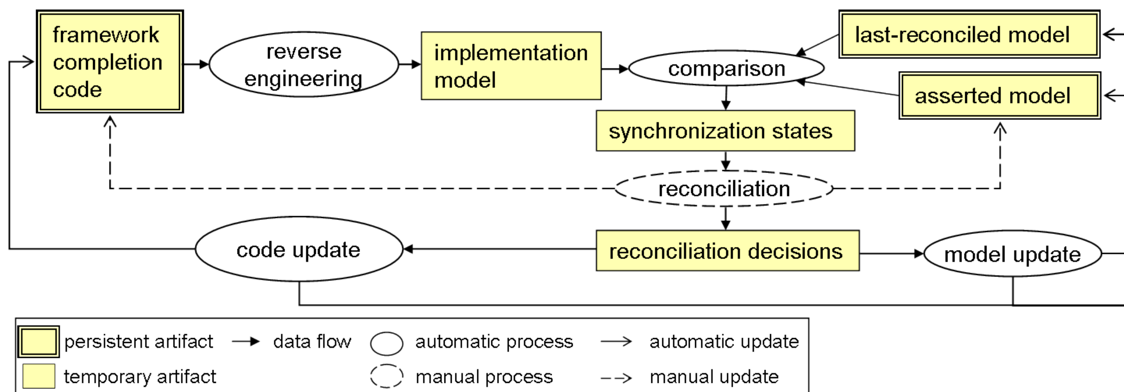

Figure 2.9: Artifacts and processes of agile round-trip engineering

Figure 2.9 presents an overview of agile round-trip engineering. The intention of agile round-trip engineering is to synchronize the current *asserted model*, which represents the intended model of the application, and the current *framework completion code*, which may be inconsistent with the asserted model. The asserted

model and the completion code that are consistent are also referred to as being *reconciled*. In order to synchronize the asserted model and the completion code, the current *implementation model* is automatically reverse engineered from the current code. Furthermore, we assume that the *last reconciled model* contains the latest copy of each feature instance that was archived after the features's most recent synchronization. Special cases occur if any of the three artifacts, namely the asserted model, the last reconciled model, or the completion code, are missing. These cases include situations in which the code has to be first created from an existing model or the model has to be first created from existing code or independently created model and code need to be synchronized for the first time.

After reverse engineering the implementation model, the three models are compared and the result of the comparison is presented to the developer. Synchronization states, the result of the comparison, describe the identified changes such as feature additions and removals or feature attribute value changes. Conflicting changes, such as attribute values that changed inconsistently, are also detected. During the reconciliation, the developer decides in which way a given change should be propagated: to the model or to the code. Reconciliation decisions towards code trigger code transformations and decisions towards model trigger model update with the features and values from the implementation model. In the following two sections we describe the comparison and reconciliation phases and the reconciliation algorithm.

Comparison

The comparison of the asserted, implementation, and last-reconciled models consists of two phases: *matching* and *computation of synchronization states*. Matching computes a tree of triples (3-tuples) $< a, i, l >$ in which features a, i, and l reside in the three models, respectively, and they correspond to each other. The tree of triples is created by overlaying trees of features from the three models. We use "-" to indicate a missing feature in a model corresponding to the position in the triple, that is, the first position corresponds to the asserted model, second to the implementation model, and the third to the last reconciled model. In our implementation we use a form of structural matching [11] in which the correspondence between features from the three models is established based on their position in each model and the values of some of their subfeatures. We encode structural information of each feature in a *key*. The computation of the key for a given feature is guided by *key annotations* attached to features: `key`, `parentKey`, and `indexKey`, and it proceeds as follows. The key always includes the name of the feature. The annotation `parentKey` specifies that the key for the given feature should include the key of the feature's parent. The annotation `indexKey` is used for multiple features and it specifies that the key for the given feature should include the feature's position in the sequence of instances of the multiple feature. Finally, the annotation `key` is used to specify the subfeatures of the given feature whose values should be included in the key. Note that it is the responsibility of FSML designer to attach key

annotations to features so that feature instances can be unambiguously identified.

Figure 2.10 presents key annotations for the fragment of the example feature model from Table 2.4. The annotations are surrounded with "|" (bar). According

```
Applet
  [1..1] name (String) |key|
 ![1..1] extendsApplet |parentKey|
    [0..1] extendsJApplet |parentKey|
  ...
  [0..*] parameter |parentKey,indexKey|
    [0..*] name (String)
```

Figure 2.10: Feature model for the concept Applet including key annotations

to the annotations, the key for an instance of the concept `Applet` will include the value of the feature `name`; the key for an instance of the feature `extendsApplet` will include the key for its parent; and the key for an instance of the feature `parameter` will include the key for its parent and the index of the instance in the sequence of parameters.

After matching, synchronization states can be computed for each triple. The computation follows a decision table in Table 2.6. The table assumes the existence of feature equality operator "=". Two features are considered equal if their values are equal or all of their subfeatures are equal. A feature can be characterized as *unchanged*, *added*, *removed*, or *modified*. A feature is modified if its value changed or any of its subfeatures changed. Additionally, a change can be *consistent* or *inconsistent*. Inconsistent (conflicting) changes propagate upwards in the feature hierarchy. For example, if a subfeature was added inconsistently then all its parent features will be added or modified inconsistently.

Reconciliation

After detecting changes for each triple, actions can be taken to reestablish the consistency between the model and the code. The reconciliation can be both manual and automatic as indicated in Figure 2.9. In manual reconciliation, developers simply update the model and the code. In automatic reconciliation, developers make *reconciliation decisions* which trigger code transformations and model updates. Possible reconciliation decisions are *enforce*, *override-and-enforce*, *update*, *override-and-update*, and *ignore*. Enforce decisions trigger code transformations and update decisions trigger model updates.

For example, if a feature was modified in the code (cf. Table 2.6, row 4), the default reconciliation decision is model update. The developer may decide on *update* or *override-and-enforce*. In the first case, the model would be updated according to the code. In the second case, the code would be overridden according to the model. Developers can also resolve conflicts using the *override-and-enforce* and *override-and-update* decisions. For example, if a feature was removed from the code and

Table 2.6: Results of three-way compare of the corresponding features $< a, i, l >$ from the asserted model A, the implementation model I, and the last-reconciled model L, respectively. The absence of a corresponding feature is represented by -. Table adapted from [10].

#	**A**	**I**	**L**	**condition**	**detected changes**
1	a	i	l	$a = i = l$	unchanged
2	a	i	l	$a = i \wedge a \neq l$	modified consistently in A & I
3	a	i	-	$a = i$	added consistently to A & I
4	a	i	l	$a \neq i \wedge a = l$	modified in I
5	a	i	l	$a \neq i \wedge i = l$	modified in A
6	a	i	l	$a \neq i \neq l \neq a$	(conflict) modified inconsistently in A & I
7	a	i	-	$a \neq i$	(conflict) added inconsistently to A & I
8	a	-	l	$a = l$	removed from I
9	a	-	l	$a \neq l$	(conflict) removed from I, modified in A
10	a	-	-		added to A
11	-	i	l	$i = l$	removed from A
12	-	i	l	$i \neq l$	(conflict) removed from A, modified in I
13	-	i	-		added to I
14	-	-	l		removed from A & I

modified in the model (cf. Table 2.6, row 9), the conflict can be resolved by adding the feature back to the code according to the model using *override-and-enforce* or by removing the feature from the model using *override-and-update*.

Reconciliation algorithm

Reconciliation algorithm executes code transformations and model updates according to synchronization states and reconciliation decisions. It is similar to the algorithm for forward engineering; however, it is driven by synchronization states and reconciliation decisions rather then just the model. The algorithm simply traverses the tree of triples in the depth-first manner starting from the feature it was invoked on and executes code transformations for enforce decisions and model updates for update decisions. The algorithm can be invoked on any triple in which at least one feature exists or on a triple whose parent triple has at least one existing feature.

In our implementation, not every change can be automatically propagated to the code. In particular, code transformations for only a few mapping types support code pattern modification (refactoring) and none support pattern removal. However, most of them support pattern addition. We evaluate round-trip engineering in Chapter 6.

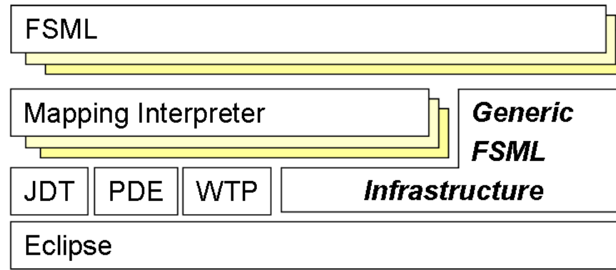

Figure 2.11: Architecture of the generic FSML infrastructure

2.6 The Generic FSML Infrastructure

The generic FSML infrastructure is a framework for implementing FSMLs. Figure 2.11 presents an architectural overview of the infrastructure. The infrastructure implements the metamodel interpretation algorithms presented in the previous sections. The infrastructure is implemented as an Eclipse plug-in and it contributes two views: Model-Code Navigation and Model-Code Synchronization, and two actions: Model-Code Synchronize and Model-Code Reconcile. The first view displays traceability links established for a selected model element in a model editor; following the link highlights a corresponding fragment of the code. The Model-Code Synchronize action executes reverse engineering and comparison processes of round-trip engineering and displays the results (synchronization states) in the Model-Code Synchronization view. The Model-Code Reconcile action executes the reconciliation algorithm for the selected synchronization triple.

The infrastructure also provides a basis for implementing new model wizards, which execute reverse engineering and create a model file. A new model wizard is implemented by extending the basic FSML Analysis Wizard.

The infrastructure also provides a support for implementing the mapping interpreters for different artifact types. Each mapping interpreter is implemented in a separate Eclipse plug-in. Each mapping interpreter has to be registered with the infrastructure before it is used; usually each FSML extends the basic Model-Code Synchronize action and it registers the required interpreters. FSMLs also register the mapping interpreters in new model wizards because they are needed for reverse engineering.

Finally, the infrastructure contributes a preferences page offering different logging options, such as computing reverse engineering statistics, enabling code variant analysis, and computing code pattern scattering metrics.

All mapping interpreters also use different Eclipse APIs. The Java mapping interpreter uses Eclipse Java Development Tools (JDT) [40] API and it relies on the parser, abstract syntax trees API, and the incremental compiler. The Java mapping interpreter handles both plain Java as well as Java projects based on the OSGi [77] component framework. Handling of OSGi components (bundles) is critical for being able to analyze Eclipse plug-ins, which are OSGi components. The Plug-in map-

ping interpreter uses Eclipse Plug-in Development Environment (PDE) [41] API for handling the plugin.xml files. Finally, the XML mapping interpreter uses Eclipse Web Tools Platform (WTP) [42] API for handling XML files.

2.7 Summary

In this chapter we presented the key ideas underlying framework-specific modeling languages. FSMLs formalize framework API concepts and can be used for expressing framework-specific models that describe how an application instantiates the concepts in the scope of an FSML. We showed how feature models can be used for modeling API concepts and how feature configurations can be used for representing framework-specific models. We described that feature instances represent code patterns in the application code and that correspondence is defined by attaching mapping definitions to features. Mapping definitions use mapping types that are generic and reusable kinds of feature-to-code-pattern correspondences. Mapping types are implemented by code queries and transformations that are used in reverse, forward, and round-trip engineering. Finally, we briefly introduced the generic FSML infrastructure that is a framework for building new FSMLs.

In the next chapter, we present a method for engineering new FSMLs. The method was extracted post-mortem from our experience with building four FSMLs.

Chapter 3

Method for Engineering FSMLs

In this chapter we present a method for engineering new framework-specific modeling languages that we devised post-mortem from the experience of building four such languages. We present the method instead of simply describing the four exemplar FSMLs because this allows us to expose the rationale behind certain design decisions we made and to present the value propositions, use cases, and interesting metamodel fragments of the FSMLs as examples. Full metamodels are presented in Appendix F.

Since the method embodies our experience with FSML engineering, we make no claims of the method's completeness or usefulness. The method presented here is our first attempt to creating an FSML engineering method and we expect it to be extended and refined in the future. Also, we leave the evaluation of the method for future work. Such an evaluation would involve an experiment in which subjects without prior FSML engineering experience would apply the method and provide feedback about correctness, completeness, and usefulness of the method.

The method focuses only on the abstract syntax and semantics of the language. Concrete syntax is outside of the scope of this thesis; however, certain information, such as, whether a concept is component- or connector-oriented, could be used in the development of the concrete syntax.

Finally, we believe that presenting an engineering method is more valuable then presenting the four FSMLs because the method is a result of the analysis of the languages that would have to be otherwise done by anybody attempting to build a new FSML.

3.1 Overview

The design and implementation of FSMLs is an iterative and incremental process. In each iteration, new entities, such as concepts, features, mapping types and definitions, and implementations of code queries and transformations are added and existing ones are evolved.

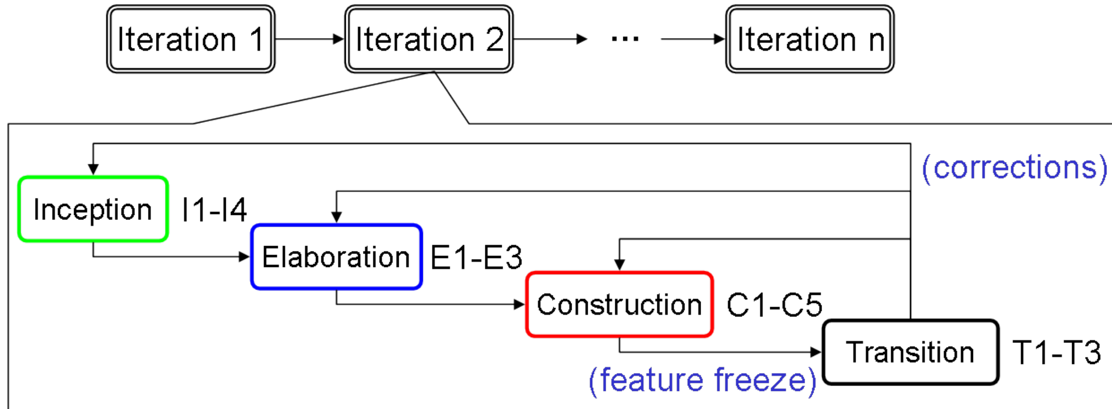

Figure 3.1: FSML life cycle: iterations and phases

Our engineering method was inspired by the Unified Process for software development [58]. The life cycle of an FSML consists of multiple iterations, each having the following phases as shown on Figure 3.1. Each iteration has different focus and requirements.

1. *Inception* is the initial planning phase. It is focused on resolving global questions such as determining the value proposition of the language and the use cases to be supported.

2. *Elaboration* is focused on identifying the main concepts and creating the overall structure of the feature model.

3. *Construction* is a refinement phase in which the concepts are decomposed into features and mapping definitions are created. Additionally, new mapping types, code queries, and code transformations can be added or existing ones can be refined.

4. *Transition* is focused on improving the quality of the language, verifying whether the language delivers its value proposition, and preparing it for the release to the users (application developers).

Within each phase, FSML developers perform several *activities* and activities consist of *steps*. Certain activities are dominant in certain phases, but each activity can potentially be performed in any phase. For example, value proposition is usually defined in the inception phase, but it can also be redefined in the subsequent phases. Furthermore, activities are performed iteratively. For example, the identification of a concept is followed by the identification of its features, which may be followed again by the identification of another concept.

The following sections describe the phases and their dominant activities. The activities and steps are presented in the form of instructions that can be directly followed by FSML developers. The presentation uses a naming scheme whereby

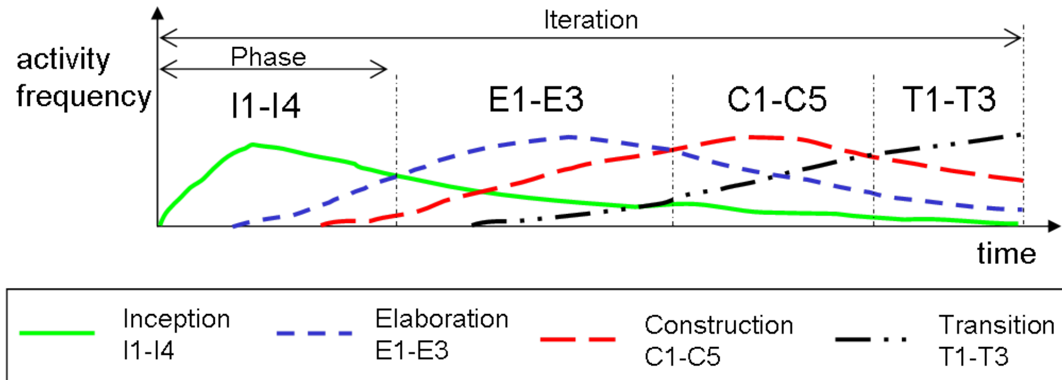

Figure 3.2: FSML life cycle: phases and their dominant activities in a single iteration

each activity is labeled with the initial letter of the corresponding phase name and its consecutive number in the activity list for the phase. Each step is labeled by its activity label and the number representing its position within its activity. For example, I1 is the label of the first activity of the inception phase and I1.1 is the first step of that activity. Figure 3.2 illustrates the frequency of performing certain activities in certain phases of the FSML's life cycle.

3.2 Inception

The input to this phase is a concrete framework for which an FSML is to be designed. Table 3.1 shows information related to the inception of the four exemplar languages. In this phase, the following activities are performed.

I1: Determine the purpose of the language. The first activity in FSML engineering is determining the value proposition for the language in the current iteration. What problem or problems should the language address? Table 3.1 summarizes the value propositions for the four exemplar FSMLs.

The main value proposition for developing WPI was that implementing Eclipse part interactions involves several implementation steps that are usually highly scattered in the completion code. As a result, determining the presence and the properties of interactions in the completion code is challenging for application developers. Also, some necessary implementation steps may easily be missed when implementing interactions. Consequently, the purpose of the WPI FSML is to ease the understanding of the interactions by providing a model and navigability to the corresponding code fragments and to simplify the creation of interactions by checking API constraints and offering round-trip engineering.

The Struts FSML was developed initially to demonstrate the feasibility of migrating Struts applications to Java Server Faces (JSF) framework. Struts and JSF are two frameworks for developing web applications that are conceptually similar,

with JSF providing more advanced features. Later (in the next iteration), the focus of the Struts FSML shifted towards visualizing page flow and checking the referential integrity between Java completion code and the corresponding XML configuration files. The latter aspect is important in Struts-based development since framework concepts are represented both in Java and XML and related by name and naming mistakes are frequent and typically discovered only at runtime. The FSML allows detecting such mistakes at development time.

The Applet FSML was developed primarily as an pedagogical example. It provides an overview of the main features of applets from the viewpoint of the Applet API and supports full round-trip engineering. The language was the main example used when extending our FSML implementation infrastructure with support for declarative mapping definitions. It was also used to compare the FSML approach to the design fragment approach, which documents usage patterns for framework APIs [46]. We used the same set of 56 applets as a benchmark for testing the FSML. Since the Applet API is relatively simple, the Applet FSML is particularly useful as an introductory example when learning the FSML approach.

The main motivation for the EJB FSML was the introduction of Java annotations as an additional configuration mechanism in EJB 3.0, which supplements the usual configuration through XML-based deployment descriptors. The EJB 3.0 specification defines a set of complex rules governing the merge and overriding of configuration information specified in Java and XML. As a result, tool support is required in order to see the final configuration and understand how it originated.

Table 3.1: Inception of the exemplar FSMLs

	WPI FSML	**Struts FSML**	**Applet FSML**	**EJB FSML**
Inception	Fall 2005	Fall 2006	Fall 2006	Fall 2007
Developers	Michał Antkiewicz	Michał Antkiewicz, Aseem P. Cheema	Michał Antkiewicz	Matthew Stephan
Framework	Eclipse Workbench 3.x	Apache Struts 1.x	Java 5.0 Applet	Sun EJB 3.0
Value proposition	Provide (1) design-level overview of the system, which contains workbench parts and interactions among them; (2) navigability to the crosscutting implementation; and (3) API constraints checking. Demonstrate feasibility of round-trip engineering.	Initial focus was to provide (1) semi-automatic completion code migration from Struts to Java Server Faces framework; later the focus changed to supporting (2) referential integrity checking between Java and XML and (3) visualizing web-page flow	(1) Drive the extension of the generic FSML infrastructure to support defining FSMLs declaratively; (2) Compare FSML approach to design fragments approach using applets as benchmark. (3) Demonstrate feasibility of round-trip engineering. (4) Provide an introductory FSML example	Provide (1) an overall configuration view merging information from Java and deployment descriptors and (2) support detection of EJB antipatterns
Use cases	RE, ACC, FE, RTE	M, later RE and RIC	AU, RE, ACC, FE, RTE	RE, RIC, ACC, MI
Horizontal scope	editors, views, selection interactions, part life cycle interactions, adapter interactions	actions, forwards, messages, forms, XML declarations	applets, status, mouse listeners, threads, parameters	EJBs, business interfaces, Java annotations, XML declarations, override rules
Artifact types	Java, plugin.xml	Java, XML	Java	Java, XML

Table 3.2: Overview of the Use Cases

Use Case	Abbrv	Description	Users
Framework API Understanding	AU	Learn API concepts and their implementation variants from FSML concept definitions	Application developers, framework instructors
Completion Code Analysis and Understanding		Use framework-specific model as a view of the system from the viewpoint defined in an FSML. Navigate from features to the implementation of the features and vice versa.	Application developers, architects, testers, and analysts
Reverse Engineering	RE	Identify concept instances in the code and present them in the form of a framework-specific model	Application developers and architects
Referential Integrity Checking	RIC	Use the extracted model to evaluate referential integrity constraints	Application developers, architects, and testers
API constraint checking	ACC	Use the extracted model to evaluate API constraints encoded in the feature model and additional constrains	Application developers, architects, and testers
Model Interpretation	MI	Analyze or transform the extracted model in order to retrieve information needed to achieve the language's value proposition. May range from simple model filtering to applying inference rules	Application architects, testers, and analysts
Forward Engineering	FE	Generate framework completion code that implements concept instances specified in the model or add implementation of new concepts and features to existing code.	Application developers and architects
Completion Code Evolution		Support evolution of the completion code by comparing models extracted at different times of the life cycle	Application developers and architects
Round-Trip Engineering	RTE	Compare and synchronize independently modified model and code	Application developers and architects
Migration	M	Migrate the completion code either to the new version of the API or to a different but conceptually similar framework	Migration engineers

The EJB FSML offers exactly such a tool. Furthermore, a secondary purpose of the FSML is to detect typical EJB antipatterns, such as *Bloated Session* or *Fragile Links* [38].

Table 3.1 additionally indicates the inception date in the first iteration of the languages.

I2: Determine which use cases should be supported.

The use cases to be supported should be the ones required to achieve the language's purpose in the current iteration. Table 3.1 lists the use cases supported by each exemplar FSML. The use cases are explained in Table 3.2.

Framework API understanding (AU) refers to the application developers and framework educators using the FSML definition to learn API concepts and their implementation variants. Since the Applet FSML covers most of the scope of the Applet API, it can be used to learn about Java applets. The remaining FSMLs focus on specific aspects of the corresponding framework API, such as areas that are difficult to understand in completion code. Although they also can be used to learn about the specific API aspects, their primary goal is supporting completion code analysis and evolution.

Completion code analysis and understanding groups four use cases concerned with analyzing and understanding completion code.

Reverse engineering (RE) is focused on identifying concept instances in the completion code and presenting them in the form of a framework-specific model. Code queries are used to match code patterns according to mapping definitions of features. If a pattern is matched, a feature is asserted to be present in feature configuration and a traceability link is established, otherwise a feature is missing. Traceability links allow navigating from features in the model to code patterns and vice versa. As a result, concept instances and their features are identified in the completion code and they are presented to the developer in a form of a framework-specific model. All four exemplars support RE.

Referential integrity checking (RIC) is focused on checking the referential integrity among concepts within one or more artifacts. The checking is performed by evaluating referential integrity constraints over the extracted model. The constraints are expressed as model queries using the pre-defined model query types from Table A.3. Struts FSML contains features that correspond to information from Java and XML and features that correspond to the referential integrity constraints.

API constraint checking (ACC) goes beyond RIC and also checks more complex constraints defined in the feature models, including the additional constraints. All exemplars support this use case.

Model interpretation (MI) involves analyzing or transforming the extracted model in order to retrieve information needed to achieve the language's value proposition. The processing may range from simple model filtering to applying inference rules. In EJB FSML, the processing involves applying override rules to produce a merged view of the EJB configuration.

Forward engineering (FE) involves generating code for an existing model from scratch or incrementally adding code for newly added concepts and features to existing code. The FSML infrastructure supports incremental code addition by code transformations. The forward engineering aspect of an FSMLs is specified compositionally, whereby the generated implementation of the model is a composition of the implementations of the features. Code transformation of each feature is executed on the existing code regardless if the existing code was written manually or was generated by code transformations executed previously for other features. Code generation is only possible if the mapping types used in mapping definitions of the features from the model have code transformations implemented. WPI, Struts, and Applet FSMLs support incremental forward engineering. Struts FSML was not required to support this use case, but it does since all code transformations for the used mapping types were already implemented.

Completion Code Evolution groups two use cases.

Round-trip engineering (RTE) involves comparing independently modified model and code and reconciling them by propagating changes in one or both directions.

The model is updated by copying features from the model of the current implementation and the code is updated by executing code transformations. If the code transformations are not implemented, RTE can still be used for comparing the code and the model and incrementally updating the model. WPI and Applet FSMLs support round-trip engineering for all of their features.

Migration (M) of the completion code can be either to a new version of the same API or to a different but conceptually similar framework. In the first case, the migration requires detecting deprecated features and adapting their implementation to the new API. In the second case, the migration can be achieved by reverse engineering the code for the source framework and using specialized forward engineering to produce the code for the target framework. Both frameworks need to be close conceptually, since a single model is used. For example, both Struts and JSF frameworks have similar concepts but different ways of implementing them and therefore the completion code can be automatically migrated [26].

I3: Determine the sources of knowledge about the framework and the domain.

The sources of knowledge useful for concept formalization include API documentation, tutorials, articles, expert knowledge, experience, sample applications, and existing metamodels and XML schemas provided with the framework. These sources of knowledge are used with varying intensity in different phases.

Table 3.3 showcases the sources of knowledge that were used in the design of the exemplar FSMLs and the distribution of the FSML features over these sources. Each three-row-block in the table provides the number of features (second row) originating from the given source (first row). The third row contains the percentage of the features from the given source to the overall number of features of the language. A summary of features originating from all documents is presented in the column ΣDoc. Features originating from non-documentation sources are presented in columns to the right of the column ΣDoc.

We explain the specific labels of the knowledge sources in Table 3.4. For example, the feature `SelectionListener` (Figure 3.3, line 29) originated from API documentation (DW), the feature `ActionDecl` (Figure 3.4, line 15) originated from an XML Schema (SS), the feature `registersMouseListener` (Figure 3.5, line 15) originated from a tutorial (TA1), the feature `typedThread` (line 53) is implied (I), the feature `deregisters` (line 19) originated from expert knowledge (E), the feature `registersKeyListener` (line 39) originated from the analysis of example applications (EA), and the feature `local` (Figure 3.3, line 6) specifying that a workbench part is a source class of the analysed project was added by an FSML developer (X).

As can be seen from Table 3.3, the sources of knowledge and the distribution of features over those sources depend much on the framework concepts in scope, the amount of material available on the framework concepts being formalized, and the experience of the FSML developer with the framework. For example, in the case where the FSML developer is a domain expert, the majority of features will likely come from his or her own experience with little coming from documentation.

Table 3.3: Distribution of features over sources of knowledge

Framework	**Source kind**	**API**		**Tutorials**		**Articles**			**ΣDoc**	**Other (non-doc)**				**ΣF**
WPI	**Source**	**DW [44]**		**TW1 [39]**		**AW1 [86]**	**AW2 [78]**	**AW3 [54]**	**D***	**I**	**E**		**X**	**F***
	Features	11	-	6	-	7	11	6	41	14	15	-	1	71
	Percentage	15.5%		8.5%		9.9%	15.5%	8.5%	57.8%	19.7%	21.1%		1.4%	100%
Struts	**Source**	**DS [19]**	**SS [3]**	**TS1 [18]**		**AS1 [17]**			**D***	**I**			**X**	**F***
	Features	3	13	14	-	8	-	-	38	7	-	-	2	47
	Percentage	6.4%	27.7%	29.8%		17.0%			80.9%	14.9%			4.3%	100%
Applet	**Source**			**TA1 [90]**	**TA2 [91]**				**D***	**I**	**E**	**EA**		**F***
	Features	-	-	19	7	-	-	-	26	23	24	2	-	75
	Percentage			25.3%	9.3%				34.7%	30.7%	32.0%	2.7%		100%
EJB	**Source**	**DE [89, 37]**	**SE [2]**	**TE1 [59]**					**D***	**I**				**F***
	Features	21	13	27	-	-	-	-	61	12	-	-	-	73
	Percentage	28.7%	17.8%	36.9%					83.5%	16.4%				100%

Table 3.4: Legend of Features' Sources

Source	**Description**
f	Letter indicating a framework. W - Eclipse Workbench, S - Apache Struts, A - Java Applet, and E - Sun EJB3.0
Df	Features that were extracted from API documentation, such as Javadoc or other specifications, where f is the framework.
Sf	Features derived from schemas contained in the framework, where f is the framework.
Tfi	Features that came from the framework's tutorials, where f is the framework and i is the identification number for the tutorial.
Afi	Features acquired from articles, where f is the framework and i is the identification number for the article.
D*	All the features derived from any of documents.
I	An implicit feature. For example, every class has a name, so a name feature is implied.
E	Features created from expert knowledge.
EA	Features acquired by going through examples of the framework.
X	Extra features added by FSML developer.
F*	All the features contained in the FSML.

On the other hand, in the case where a FSML developer has little to no knowledge of the domain, such as in the case of the EJB FSML, the majority of features will come from documentation with little to none coming from experience or expert knowledge.

For frameworks that have extensive configuration schemas, such as Struts and EJB, the schemas were a significant source of knowledge, for example, 28% of features for Struts and 18% for EJB. Such schemas can be viewed as metamodels of the artifacts they represent and can be, to some degree, incorporated into the FSML.

I4: Determine the scope of the language.

The purpose of this activity is to determine the criteria for deciding whether a given concept or feature is in scope of the language or not. We distinguish two kinds of scope.

Horizontal scope is measured with respect to the coverage of the API. The FSML could have the full API in scope or just parts of it. The breadth of the horizontal scope can be delineated early by deciding which top-level concepts should be included. The decision is guided by the value proposition. For example, the purpose of the FSML could be to (1) enforce certain API constraints and good practices and to detect typical errors or omissions, antipatterns, or bad code smells or (2) support understanding and implementing features that are difficult to locate in the code due to scattering and tangling. For each case, only the concepts and features involved in these areas of difficulty would be included. The horizontal scope is also influenced by the kinds of stakeholders, their experience, and their viewpoints. For example, quality assurance engineers are likely to be interested in detecting API constraint violations, whereas developers are likely to have broader interests. The key concepts in scope of the exemplar languages are listed in Table 3.1.

Vertical scope refers to the depth of the feature models. The deeper the models, the more detailed features are captured. Usually, the leaves of the hierarchy correspond to the implementation steps stipulated by the API, such as calling a framework method, or the parameters of these steps, method call's arguments. The vertical scope is usually impacted by the choice of the use cases to be supported.

Languages targeting API understanding are likely to cover the entire API. Their vertical scope may emphasize the most used areas and perhaps also the more tricky ones. The vertical scope of the Applet FSML is fairly balanced. Languages for reverse engineering are likely to focus on areas that are more difficult to understand, such as viewpart interactions for WPI. Languages supporting integrity checking, e.g., Struts FSML, are quite selective in terms of their horizontal scope. Model interpretation involves adding higher level concepts for representing the results of the model analysis. Also, the extracted model is typically scoped towards the model queries that need to be executed, which is the case for the antipattern detection in the EJB FSML. Adding support for incremental forward engineering and round-trip engineering typically requires deeper vertical scope than reverse engineering. We observed this in WPI and Applet FSMLs. Adding support for these use cases

required adding more detailed features, such as those representing method call arguments. Finally, migration requires determining the commonalities and differences between the source and target domain of the migration.

Table 3.1 additionally specifies the artifact types covered by the languages.

3.3 Elaboration

In this phase, the following activities are performed.

E1: Identify framework-provided concepts in the scope. Use the identified sources of knowledge, such as,

- *API documentation, tutorials, articles.* These sources provide concepts and features and their intended use as envisioned by the framework developers.
- *Experience and best practices.* These sources provide features related to how the expert users use the framework. Note that experts often also contribute articles and tutorials.
- *Sample applications.* Analysis of sample applications, both supplied with the framework and real-life applications, provides information about the typical use of the framework as well as about other implementation variants of concepts that are not included in other sources.
- *Metamodels and XML schemas.* These sources provide concepts and features made explicit by framework developers. Note that various configuration dialogs and wizards supplied together with the framework can be based on those metamodels or schemas.

E2: Determine kinds of concepts in scope. We distinguish three main kinds of concepts.

Component-oriented concepts are concepts that usually correspond to code components, such as Java classes and XML documents.

For example, `ViewPart` (Figure 3.3, lines 7-11) is a component-oriented concept.

Connector-oriented concepts are concepts that relate other concepts with each other. Connector-oriented concepts can correspond to code components (e.g., listeners) but they can also correspond to code patterns scattered across the components related by the connector, such as method calls or XML declarations.

For example, `AdapterRequestor` (Figure 3.3, lines 66-70) is a connector-oriented concept. It relates a part referenced through the `requestor` reference (line 67) with multiple `AdapterProviders` referenced through the `adapterProvider` reference (line 70). The method call represented by the essential feature `requestsAdapter` is what actually connects a requestor with the provider.

```
1  [1..1] WorkbenchPartInteractions <project>
3    [0..*] Part <class>
4     ![1..1] name (String) <className>
5      [0..1] package (String) <qualifier>
6      [0..1] local <isLocal>
7    [0..*] ViewPart -|> Part
8      [0..1] partId (String) <viewPartId>
9     ![1..1] implementsIViewPart <assignableTo: 'IViewPart' concrete: true> <subsumedBy:
   extendsViewPart/ >
10       [0..1] extendsViewPart <assignableTo: 'ViewPart'> <subsumedBy: extendsPageBookView>
11         [0..1] extendsPageBookView <assignableTo: 'PageBookView'>
12   [0..*] EditorPart -|> Part
13     [0..1] partId (String) <editorPartId>
14     [0..1] contributor <class>
15      ![1..1] contributor (String) <editorContributor> <fullyQualifiedName>
16       [1..1] extendsEditorActionBarContributor <assignableTo: 'EditorActionBarContributor'>
   <subsumedBy: extendsMultiPageEditorActionBarContributor>
17         [0..1] extendsMultiPageEditorActionBarContributor <assignableTo: 'MultiPageEditorActionBar-
   Contributor'>
18       [1..1] multiPageEditorAndContributor <constraint ../../extendsEditorPart/extendsMultiPage-
   EditorPart implies: ../extendsEditorActionBarContributor/extendsMultiPageEditorActionBarContributor>
19    ![1..1] implementsIEditorPart <assignableTo: 'IEditorPart' concrete: true> <subsumedBy:
   extendsEditorPart/>
20       [0..1] extendsEditorPart <assignableTo: 'EditorPart'> <subsumedBy: extendsMultiPageEditorPart>
21         [0..1] extendsMultiPageEditorPart <assignableTo: 'MultiPageEditorPart'>
22      [0..0] extendsMultiPageEditor <assignableTo: 'MultiPageEditor'> <deprecated>
29   [0..*] SelectionListener <class>
32    ![1..1] registersAs
33      !<1-3>
34         [0..*] globalSelectionListener <callsTo: 'void ISelectionService.addSelectionListener(ISe-
   lectionListener)' receiverExpr: 'getSite().getPage()' >
35           [1..*] deregisters <callsTo: 'void ISelectionService.removeSelectionListener(ISelection-
   Listener)' receiverExpr: 'getSite().getPage()'>
36            ![1..1] deregistersSameObject <argument: 1 ofCall: ../ sameAsArg: 1 ofMethodCall:
   ../../>
37              [1..1] registersBeforeDeregisters <methodCall call: ../../../ before: ../../
   givenCallbackSeq: 'init createPartControl dispose'>
38         [0..*] globalPostSelectionListener <callsTo: 'void ISelectionService.addPostSelectionListe-
   ner(ISelectionListener)' receiverExpr: 'getSite().getPage()' >
42         [0..*] specificSelectionListener <callsTo: 'void ISelectionService.addSelectionListener(String,
   ISelectionListener)' receiverExpr: 'getSite().getPage()'>
62   [0..*] AdapterProvider <class>
63    ![1..1] provider (Part) <baseConcept>
64    ![1..1] providesAdapter <allMethods signature: 'Object getAdapter(Class)'>
65      ![1..*] adapters (String)  <returnedObjectTypes ifkey: 1>
66   [0..*] AdapterRequestor <class>
67    ![1..1] requestor (Part) <baseConcept>
68    ![1..*] requestsAdapter <callsTo: 'Object IAdaptable.getAdapter(Class)' receiver:
   'IWorkbenchPart' receiverExpr: 'iWorkbenchPart'>
69       [1..1] adapter (String) <valueOfArg: 1>
70       [0..*] adapterProvider (AdapterProvider) <where attribute: providesAdapter/adapters contains:
   ../adapter>
```

Figure 3.3: Excerpt of WPI FSML Metamodel

Port-oriented concepts are concepts that interface between the components and connectors, that is, they enable connecting a connector to a component.

For example, `BusinessInterface` (Figure 3.6, lines 3-12) is a port-oriented concept. The concept `BusinessInterface`, and all non-abstract concepts that inherit from it, allow EJB clients to connect to EJBs. Specifically, EJB clients connect to EJBs through a business interface by specifying the interface in Java annotations, *@Local* or *@Remote*, put on the EJB class or through the XML deployment descriptor.

```
1   [1..1] StrutsApplication <project>
3     [1..1] StrutsConfig <xmlDocument: '/WEB-INF/struts-config.xml'> <xmlElement name:
    'struts-config'>
15      [0..*] ActionDecl <xmlElements: 'action-mappings/action'> <xmlElement>
16        [1..1] path (String) <xmlAttribute>
17        [0..1] name (String) <xmlAttribute>
18        [0..1] type (String) <xmlAttribute>
19        [1..1] actionImpl (ActionImpl) <where attribute: qualifiedName equalsTo: ../type>
32    [0..*] ActionImpl <class>
35      [1..1] qualifiedName (String)
37     ![1..1] extendsAction <assignableTo: 'Action'>
```

Figure 3.4: Excerpt of Struts FSML Metamodel

The characterization of the concepts is helpful in determining the structure of the language, that is, which concepts should be main concepts and which concepts should be parts (subfeatures) of other concepts. In WPI FSML, the concept `ViewPart` was modeled as an individual concept so that it can be represented in the model once and referenced by other concepts. We also chose to model all connector-oriented concepts in WPI FSML as individual concepts; however, they could as well be modeled as subfeatures of parts. In EJB FSML, business interfaces that begin with *Explicit* are modeled as individual concepts and they are referenced (Figure 3.6, lines 40, 41). Business interfaces that begin with *Derived* are subfeatures of other concepts (lines 33, 36).

E3: Define the overall structure of the feature model.

E3.1: Introduce features for grouping related concepts. A feature model can be divided into logical parts that group related concepts/features. The concepts can be logically related or grouped based on the artifact they represent.

For example, the feature `InformationFromAnnotations` (Figure 3.6, line 2) is used for grouping subfeatures related to Java annotations and the feature `InformationFromDeploymentDescriptor` (line 60) is used for grouping subfeatures related to the XML deployment descriptor.

Subfeatures can also be logically grouped inside a single concept.

For example, the feature `overridesLifecycleMethods` (Figure 3.5, lines 6-12) is used for grouping features corresponding to life cycle methods.

E3.2: Determine the order of concepts. This step is needed only if metamodel interpretation algorithms rely on the ordering of features and the `base concept reference` mechanism is used (cf. Section 2.5.2). Base concept reference mechanism is used for defining separate concepts whose instances correspond to code patterns that other (existing) concept instances correspond to. Base concepts, that is, concepts referenced through a base concept reference, should be processed before the concepts referencing them.

For example, the feature `requestor` (Figure 3.3, line 67) is a base concept reference of the concept `AdapterRequestor` (line 66) and specifies that each instance of the concept `AdapterRequestor` will correspond to the same code pattern (class in

```
1  AppletModel <project>
2    [0..*] Applet <class>
4     ![1..1] extendsApplet <assignableTo: 'Applet' local: true> <subsumedBy: extendsJApplet>
6      [0..1] overridesLifecycleMethods
7       !<1-5>
8        [0..1] init <methods: 'void init()'>
12       [0..1] destroy <methods: 'void destroy()'>
15     [0..*] registersMouseListener <callsTo: 'void Component.addMouseListener(MouseListener)'>
16      !<1-1>
17       [0..1] this <argumentIsThis: 1>
19         [1..1] deregisters <callsTo: 'void Component.removeMouseListener(MouseListener)'>
39     [0..*] registersKeyListener <callsTo: 'void Component.addKeyListener(KeyListener)' position:
   'after' location: 'void init()'>
51     [0..*] Thread <field>
52       [1..1] thread (String) <fieldName>
53        ![1..1] typedThread <fieldOfType: 'Thread'>
54         [1..1] InitializesThread
55          !<1-1>
56            [0..1] initializesThreadWithRunnable <assignedNew: 'void Thread(Runnable)' position:
   'after' location: 'void init()'>
66            [0..1] initializesWithThreadSubclass <assignedNew: initializer: true subtypeOf:
   'Thread'>
68              [1..1] overridesRun <methods: 'void run()'>
70         [1..1] nullifiesThread <assignedNull>
71     [0..*] singleTaskThread <callsTo: 'void Thread(Runnable)' statement: true>
78     [0..*] parameter <callsReceived: 'String Applet.getParameter(String)' location: 'void init()'>
79       [0..*] name <valueOfArg: 1>
80     [0..1] providesParameterInfo <methods: 'String[][] getParameterInfo()'>
81     [1..1] providesInfoForParameters <constraint: ../parameter implies: ../providesParameterInfo>
```

Figure 3.5: Excerpt of Applet FSML Metamodel

this case) that the referenced base concept instance corresponds to (instance of the concept `Part` in this case). Also, the concept `Part` is processed before the concept `AdapterRequestor` because of their order in the feature model.

3.4 Construction

In this phase, the feature model is refined and the required elements of the approach are created.

C1: Decompose the concepts into features.

C1.1: Choose appropriate level of abstraction and granularity. Which features are important with respect to the value proposition? Which features can be abstracted away? Below we list some criteria on deciding which features are potentially useful to include.

Features corresponding to implementation steps. Such features correspond to steps, such as, subclassing a framework class, implementing a framework interface, implementing callback methods, invoking a framework service, placing a Java annotation, or creating framework-stipulated XML declarations and setting XML attribute values.

Most of the features correspond to implementation steps. For example, in order to create an instance of the concept `Applet` (Figure 3.5), the developer must create

```
1  EJBProject <project>
2    [0..1] InformationFromAnnotations
3      [0..*] BusinessInterface <class>
4        [1..1] interfaceName (String) <fullyQualifiedName>
5      [0..*] DerivedLocalInterface -|> BusinessInterface
6      [0..*] DerivedRemoteInterface -|> BusinessInterface
7      [0..*] ExplicitLocalInterface -|> BusinessInterface
8       ![1..1] localAnnotation <annotatedWith: 'Local'> <annotation>
9         ![1..1] isMarker <hasNoAttribute>
10     [0..*] ExplicitRemoteInterface -|> BusinessInterface
11      ![1..1] remoteAnnotation <annotatedWith: 'Remote'> <annotation>
12        ![1..1] isMarker <hasNoAttribute>
13     [0..*] EJBClass <class>
31     [0..*] SessionBean -|> EJBClass
32       [0..1] localInterfaceSpecification <annotatedWith: 'Local'> <annotation>
33        ![1..*] localInterfaces (DerivedLocalInterface) <attribute: 'value'>
35       [0..1] remoteInterfaceSpecification <annotatedWith: 'Remote'> <annotation>
36        ![1..*] remoteInterfaces (RemoteLocalInterface) <attribute: 'value'>
37           [1..1] interfaceName (String) <fullyQualifiedName>
38       [0..*] implementedLocalInterface <ImplementsExplicitLocalInterface>
39       [0..*] implementedRemoteInterface <ImplementsExplicitRemoteInterface>
40       [0..*] explicitLocalInterface (ExplicitLocalInterface) <where attribute: interfaceName in:
../implementedLocalInterface>
41       [0..*] explicitRemoteInterface (ExplicitRemoteInterface) <where attribute: interfaceName in:
../implementedRemoteInterface>
42     [0..*] StatelessEJB -|> SessionBean
43      ![1..1] statelessAnnotation <annotatedWith: 'Stateless'> <annotation>
60   [0..1] InformationFromDeploymentDescriptor <xmlDocument: '/META-INF/ejb-jar.xml'> <xmlElement
name: 'ejb-jar'>
64     [0..*] DDStatefulEJB -|> DDSessionBean <xmlElements: 'enterprise-beans/session'>
65      ![1..1] sessionType <xmlElements: 'session-type'> <xmlElement>
66        ![1..1] isStatefulSessionType <xmlElementValueEqualsString StringToSearchFor: 'Stateful'>
```

Figure 3.6: Excerpt of EJB FSML Metamodel

a Java class (line 2) which extends the framework class `Applet` (line 4). Next, the developer may override life cycle methods (lines 8-12), register and deregister mouse listeners by adding calls to the appropriate methods (lines 15, 19), or retrieve values of parameters (lines 78-79) by adding a method call and specifying the name of the parameter.

Frequently used features. Such features increase the usefulness of the language.

Examples of features that almost every applet has include the feature `init` (Figure 3.5, line 8) and `parameter` (lines 78-79).

Composite features. Composite features are features that have some structure or that are at a higher level of abstraction. These features raise the level of abstraction above the code.

For example, the feature `StrutsConfig` (Figure 3.4, line 3) is a composite feature because it groups all features related to the form, action, and forward declarations in `struts-config.xml` file.

Features involved in complex constraints or dependencies. Including these features in the language will enable checking these constraints in the model.

For example, the features `parameter` and `providesParameterInfo` (Figure 3.5, lines 78, 80) are involved in a constraint (line 81). The constraint asserts that if at least one parameter is used, the applet should provide information about the parameters.

Features with a life cycle. These features are related to API life cycle constraints such as that certain framework services must be invoked in conjunction and in a certain order. Such constraints are easy to violate in the code and therefore checking them in the model may support the developers.

In our exemplar FSMLs, features related to listeners have a life cycle because the listener has to be first registered and deregistered when no longer needed. The feature `globalSelectionListener` (Figure 3.3, line 34) corresponds to the registration method call and the feature `deregisters` (line 35) corresponds to the deregistration method call. Furthermore, the features on lines 36 and 37, specify that the same object must be used in both method calls and that the registration must occur before deregistration, respectively.

Features with variability. These are the features that can be implemented in alternative ways. The implementation variants may be equivalent whereby the same effect is achieved in different ways or they may be alternative whereby different effects are achieved for the same feature.

The feature `Thread` (Figure 3.5, lines 51-70) is an example of a feature with two equivalent variants. In Java, a thread can be initialized in two ways: by instantiating the `Thread` class (line 56) or subclassing the `Thread` class (line 66). The choice is modeled using a feature group (line 55).

The feature `registersAs` (Figure 3.3, line 32) is an example of a feature with three alternative variants. A selection listener can be registered as a global selection listener (line 34), a global post selection listener (line 38), or a specific selection listener (line 42). Registering a listener in different ways achieves different effects for the same feature.

In general, there are different degrees of variability: more configuration-oriented vs. more construction-oriented. Both examples above are more configuration-oriented. The more cloning and referencing involved in concept instantiation, the more construction-oriented the variability of the concept is.

Scattered features. These are the features which correspond or whose subfeatures correspond to code patterns scattered across the completion code. Such features provide an alternative-to-code decomposition in which the related features are presented in a single hierarchy whereas the corresponding code patterns are scattered across the code.

In our exemplar FSMLs most composite features are scattered features, for example the feature `Thread` (Figure 3.5, lines 51-70).

C1.2: Decide on feature nesting. Which features should be parent features and which should be subfeatures? Feature nesting may mirror relationships between the code patterns that the features correspond to.

Code pattern nesting. Features corresponding to the nested patterns may be children of the features corresponding to the parent patterns.

For example, the features corresponding to fields and methods are children of features corresponding to classes (Figure 3.5, lines 8-12, 51, 80). Also, features

corresponding to XML attributes are children of features corresponding to XML elements (Figure 3.4, lines 16-18).

Semantic implications. In feature models, a subfeature implies its parent feature. In the code, certain patterns may imply other patterns and so the corresponding features may mirror the implication.

For example, features corresponding to being assignable to more specific types are children of features corresponding to being assignable to more general types (Figure 3.3, lines 9-11) because of the semantics of type inheritance.

Logical dependency. Certain features are meaningful (or should be considered) only in the presence of other features. In that case the former can be children of the latter.

For example, the feature `overridesRun` (Figure 3.5, line 68) specifying that a subclass of the class `Thread` should override the method `void run()` only applies if the variant with a subclass is used (line 66).

C1.3: Specify feature cardinality. What are the code patterns (both structural and behavioural) that implement the features? In general, features correspond to code patterns in the completion code stipulated by the API and their cardinality depends on the possible number of the patterns.

Essential features. These features correspond to patterns that are the minimum required to identify a concept instance. Without its essential features an instance cannot be considered an instance of a particular concept.

For example, essential features of component-oriented concepts may correspond to being assignable to a certain type (Figure 3.5, line 4 or 53), being annotated with a Java annotation of a certain type (Figure 3.6, line 43), or being declared as a component in an XML configuration file (Figure 3.6, lines 65-66). Essential features of connector-oriented concepts may correspond to method calls that attach the connector to a component (Figure 3.3, line 68). Finally, an essential feature of port-oriented concepts may correspond to explicitly naming the port in a Java annotation of the component (Figure 3.6, lines 35-37).

Mandatory features. These features correspond to patterns that should be present according to the API but which are not essential. Mandatory features have cardinality [1..1].

For example, the features `initializesThread` (Figure 3.5, line 54) and `nullifiesThread` (line 70) are required for the correct implementation but they are non-essential since only the feature `typedThread` (line 53) is required to determine that a given field is a thread.

Often, mandatory features are used to model constraints, so that a missing mandatory feature indicates constraint violation.

For example, the feature `providesInfoForParameters` (line 81) corresponds to an API constraint that should always be satisfied.

Optional features. These features correspond to patterns that may be present according to the API. Optional features have cardinality [0..1]. Optional features can be grouped in feature groups to capture certain constraints, such as, that certain patterns are exclusive or alternative.

For example, the features on lines 10 and 11 in Figure 3.3 indicate that the developer can optionally extend two other API classes. We have already seen examples of optional grouped features (Figure 3.5, lines 56 and 66).

Multiple features. These features correspond to patterns that can be repeated in the code. We distinguish three kinds of multiple features depending on the lower bound of the feature cardinality: optional multiple with cardinality [0..n], mandatory multiple with cardinality [m..n], and essential multiple with cardinality ![m..n], where $m, n > 1 \wedge (n > m \vee n = *)$.

For example, the feature on line 34 in Figure 3.3 corresponds to the registration method calls and it is an optional multiple feature. In the feature configuration, each clone (instance) of the feature will correspond to a single method call. The feature on line 35 is a mandatory multiple feature because for every registration method call there should be at least one deregistration call. The feature on line 68 is an essential multiple feature because a given part is an adapter requestor only if it requests at least one adapter by calling the method `getAdapter`.

Prohibited features. These features correspond to patterns that should not be present in the code. Prohibited features have cardinality [0..0]. Prohibited features may indicate common problems or API misuses as well as uses of the deprecated API.

For example, the feature `extendsMultiPageEditor` (Figure 3.3, line 22) corresponds to extending a deprecated API class `MultiPageEditor`.

C1.4: Specify additional constraints. Feature hierarchy cannot capture certain constraints involving features from different parts of the hierarchy. However, certain constraints can be specified by adding mandatory features as shown is step C1.3.

Referential integrity constraints can also be expressed by mandatory reference features, i.e., features with an attribute which can point to other features. Referential integrity constraints are constraints of the form $\forall x.\exists y | x.a = y.b$ or $\exists y | y.v = x$ for the given x.

For example, the feature `actionImpl` (Figure 3.4, line 19) corresponds to a referential integrity constraint that action implementation must exist for every action declaration. `actionImpl` is a reference feature and can point to instances of the concept `ActionImpl` (line 32). The constraint specifies that the value of the feature `qualifiedName` of the `ActionImpl` must be the same as the value of the feature `type` of `ActionDecl`. Table A.3 shows constraints used in the exemplar FSMLs.

C1.5: Define reference features. Connector-oriented concepts relate a number of other concepts and therefore they need to point to concept instances in the model. Reference features can be used for that purpose, since values of reference attributes are other features.

In activity E2, we presented an example of a connector-oriented concept `AdapterRequestor` which references adapter providers through the `adapterProvider` reference feature (Figure 3.3, line 70). The constraint specifies that the values of that reference should be all instances of the `AdapterProvider` concept whose list of provided adapters contains the requested adapter.

C2: Create mapping definitions for the features. Mapping definitions specify the correspondence between features and code patterns.

C2.1: Choose a mapping type. Choose an existing mapping type that defines the correspondence of the feature to code patterns. Define a new mapping type if needed.

We present the mapping types used in this thesis in Tables 4.1-4.2 and Tables A.1-A.2. In the following description, we use abbreviations from the third column of these tables to refer to the mapping types.

Each mapping type can have a number of parameters. We categorize the parameters into *core*, *reverse*, and *forward* parameters. Core parameters define the correspondence, reverse parameters are used only by code queries to control code pattern matching, and forward parameters are used only by code transformations to control code pattern creation. Core and reverse parameters are included in Tables 4.1-4.2 and Tables A.1-A.2. Forward parameters are presented in Table A.4 since they are common for many mapping types. Some parameters are optional and we present them in square brackets. Furthermore, we categorize the parameters into *static* and *dynamic*.

C2.2: Set the values of the static parameters in the mapping definition. Extend the mapping with new parameters as needed.

For example, the forward parameter `receiverExpr` was added to specify the expressions that are generated as receivers of method calls. The parameter can be used in conjunction with the mapping types `callsTo` and `callsRec`. For example, the method call for the feature `globalSelectionListener` (Figure 3.3, line 34) will be generated as `getSite().getPage().addSelectionListener(null)`.

C2.3: Specify features for the retrieval of dynamic parameter values. For each dynamic parameter, decide whether the value should be determined implicitly using the context mechanism or explicitly using a path. Make sure that parent features exist for the parameters whose value will be retrieved using the context mechanism.

- Adjust the order of features to make sure that the required feature will have the value before the given mapping can be evaluated,
- Insert new parent features to be contexts as required by the mapping,
- Define paths to other features that have the required values.

For example, features on lines 34-37 in Figure 3.3 use both the context mechanism and explicit paths. The mapping type `callsTo` requires a Java class as a value of

its parameter `c`. The parameter is left unspecified for the features on lines 34 and 35 and therefore the context mechanism will be used to retrieve the value of the parameter. The mapping type `argSameObj` requires two method calls as values of its parameters `mc1` and `mc2`. In the mapping definition on line 36, path expressions are used to explicitly point to the features from which the method calls should be retrieved. Analogously explicit paths are used in the mapping definition for the feature on line 37.

C2.4: Implement missing code queries and transformations for the used mapping types if required by the chosen use cases.

For example, currently code transformations for mapping types related to Java annotations (`annotatedWith`, `attribute`, and `noAttribute`) are not implemented because the EJB FSML, which uses them, does not need to support the forward engineering use case. All mapping types used in WPI and Applet FSMLs have code transformations implemented since the languages must support forward engineering and round-trip engineering use cases.

C3: Put key annotations. Key annotations are required for round-trip engineering because the correspondence between the features from the three models is established based on their keys. Keys are also needed for establishing traceability links during reverse engineering. For each feature put the annotation `parentKey` if the key of the parent should be included in the feature's key. For multiple features that cannot be distinguished based on their subfeatures, put the annotation `indexKey` to include the index of the given feature in the sequence of instances which are children of the same parent feature. Put the annotation `key` on the subfeatures of the given feature whose values should be included in the key of the given feature. Figure 2.10 illustrates a concrete example.

The annotation `indexKey` should be used as rarely as possible because it makes the key very sensitive to the position of the given feature in the list of instances which easily changes when the code is rearranged.

C4: Add support for new use cases. Usually, the evolution of mapping types, code queries, and transformations is driven by the need to achieve the value proposition of a language. Often, however, it is possible to add support for new use cases with very little cost.

For example, initially Struts FSML was not required to support round-trip engineering. However, since all required code transformations for the mapping types used in the language are available, adding support for round-trip engineering amounts to setting forward parameters in mapping definitions.

C5: Build a test suite. A test suite is the completion code that includes features supported by the language. It can be used to test the use cases.

The code should have both correct and incorrect patterns. The former should be matched by code queries and produced by code transformations. The latter should be missed by code queries.

Feature models typically have very large number of possible configurations and it is not possible to build code that tests each configuration. However, there is some degree of orthogonality in the feature models, meaning that not all features are related with each other. This facilitates testing parts of the feature model separately, thus greatly reducing the number of configurations that have to be tested.

3.5 Transition

In this phase, the language is extensively tested and refined until the required quality and usefulness are achieved. In this phase, no more new concepts and features are added (i.e., *feature freeze* in Figure 3.1); however, modifications to the metamodel required for fixing errors are allowed (i.e., *corrections* in Figure 3.1). Below we list activities that may be performed in this phase.

T1: Perform system testing. System testing aims at answering the following questions. Is the structure of the feature model correct? Are the mapping definitions correct? Is the implementation of code queries and the code transformations correct? What is the precision and recall of the code queries? Does the generated code compile? Is it complete or skeletal? Are the code patterns created by code transformations matched by code queries?

T1.1: Test the structure of the feature model. Check satisfiability, that is, whether the set of legal feature configurations is non-empty. Create example feature configurations and verify that meaningful configurations are allowed by the feature model and additional constraints and that incorrect configurations are disallowed. Correct feature nesting (cf. C1.2), cardinality constraints (cf. C1.3), and additional constraints (cf. C1.4).

T1.2: Verify mapping definitions. Verify that the metamodel is well-typed (cf. Section 2.5), that is, (1) static mapping type parameters are set and their values are correct (e.g., correct method signatures or type names), (2) target features specified by path expressions exist and have a required mapping type, and (3) parent features with the required mapping type exist if the context mechanism is used. Correct feature nesting (cf. C1.2), mapping definitions (cf. C2), feature ordering (cf. C2.3).

T1.3: Test the implementation of code queries and transformations. Refine code queries and transformations to improve the quality of the retrieved models and the generated code. Adjust the feature model to the capabilities of the queries and transformations.

We first developed simple code queries for Java that employed simple approximations of behavioural code patterns. We then evaluated the precision and recall of those code queries [13]. Although the precision and recall were very high, we learned that the code queries can be improved without incurring a prohibitive increase in the execution time. We also implemented and evaluated the precision and recall of the new and more sophisticated code queries (cf. Section 4).

However, the feature models had to be adjusted to support the more powerful code queries. For example, a simple code query for the mapping type `argVal` was returning a value of a method call argument only if the argument was a constant or a literal. Therefore it was sufficient for the feature `name` (Figure 3.5, line 79) to have the cardinality [0..1]. The new code query performs constant propagation if the argument of a method call is a variable. Because constant propagation may return multiple potential values of the variable, the cardinality of the feature `name` had to be changed to [0..*].

T1.4: Refine the test suite. As the feature models, mapping types, code queries, and code transformations evolve, the test suite should be extended to cover new features and implementation variants.

T2: Perform acceptance testing. Is the language fulfilling its purpose and providing the value as expected? Are the required use cases supported?

T2.1: Include missing implementation variants. Use API, sample applications, and best practices to identify missing implementation variants. Extend the feature models, mapping definitions, mapping types, code queries, and code transformations as needed.

During the evaluation of the precision and recall of code queries we analyzed large body of sample applications and we noticed few additional variants for implementing features that were not included in the FSMLs.

For example, applet threads can be also initialized by subclassing the class `Thread` and directly overriding the method `run` (this variant is also explicitly mentioned in the API). We refined the Applet FSML by inserting a feature group (Figure 3.5, line 55) and adding new features (lines 66-68).

We also noticed in the sample applications that some applets used single-task threads that were not assigned to a field but were simply instantiated in an individual statement (regular threads are instantiated in the right hand side of an assignment to a field). To support single-task threads, we added new features (starting at line 71). We also had to extend the mapping type `callsTo` by introducing a new parameter `statement` that specifies whether only method calls that are statements should be matched by the code query.

Similarly, we refined code transformations to enable generating different variants. During code generation, certain additional decisions have to be made and these are specified using the forward parameters of the mapping types.

For example, the code transformation for the mapping type `assignNew` creates an assignment to a field in which the right hand side is a constructor call of the given signature. Such an assignment can be created in two ways: as an individual statement in a method body or as a field initializer. Mapping definition of the feature on line 56 uses the first variant and the assignment will be generated at the end of the method `init`. Mapping definition of the feature on line 66 uses the second variant by setting the value of the forward parameter `initializer` to `true`.

T2.2: Test support for the required use cases. Verify that the language can be used to perform the required use cases.

For example, in Chapters 5 and 6 we present tests we performed to verify that Applet and WPI FSMLs support the forward and round-trip engineering use cases.

T3: Perform beta testing. Beta testing aims at answering the following questions. Is the retrieved model useful for application developers? Is the generated code useful for application developers? How much more work do application developers need to invest to get the code working?

None of our sample languages have been released to the public yet; however, we performed the evaluation of code queries on real-life applications [13] (cf. Chapter 4) and we performed tests of the forward engineering and round-trip engineering of Applet and WPI FSMLs, as described later in Chapters 5 and 6. We think that the evaluated languages would provide value to application developers.

The purpose of this phase is to deliver a working end product of the iteration. The phase and the iteration end when the language can deliver the value proposition established in the inception phase. Chapters 4–6 present tests that can be performed to evaluate reverse, forward, and round-trip engineering capabilities of the language. Finally, when the constructed FSML delivers the value proposition of this iteration, a new iteration may begin.

3.6 Summary

In this chapter we presented an FSML engineering method extracted post-mortem from our experience in building four FSMLs. The method is a result of the analysis of the exemplar languages. The method divides the life cycle of an FSML into iterations, each iteration into phases, each phase into activities, and each activity into steps. We presented the phases, activities, and steps that guide an FSML developer in each iteration and we illustrated them with examples from the four FSMLs. The next three chapters present the evaluation of reverse, forward, and round-trip engineering. Chapter 7 presents the evaluation of the method.

Chapter 4

Evaluation of Reverse Engineering

4.1 Introduction

In this chapter we report on a study 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 simple code queries and proposing the refined code queries, and
3. evaluation of the precision and recall of the refined code queries.

In the first phase, we analyzed mapping definitions attached to features in three exemplar *framework-specific modeling languages* (FSMLs). The FSMLs used in this study were designed for: (i) Java Applet framework [88], (ii) Apache Struts framework [16], and (iii) a part of the Eclipse Workbench framework [78]. 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 [13].

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.

The main contribution of this chapter 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), simple code queries are often sufficient for reverse engineering. 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.

4.2 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 using *callback methods*. Due to the inversion of control, both the application and the framework need to be considered during the analysis. Also, frameworks commonly interpret configuration files and use reflection to dynamically load and instantiate application classes. Therefore, the construction of the complete and precise control flow graph, which is a basis for many other static analyses, is often 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 spec-

ifying 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 [4] or a method can be assigned the role of an *action method* in Java Server Faces [87] 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 [45] 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 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.

4.3 Setup of the Study

We conducted the study in three phases. The purpose of the first phase was to identify 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.

4.3.1 Setup of phase 1

The inputs to the first phase of the study are three exemplar FSMLs, one for each of the following frameworks: Java Applet [88], Apache Struts [16], and a

part of Eclipse Workbench [78]. 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 4.4.

4.3.2 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 abstract 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 [46]. 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 [15] (v.3.0), Mailreader [16] (v.1.3.8), and Cookbook [16] (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 4.4 and the data relative to their precision and recall is described in Section 4.5.1. The results are discussed in Sections 4.5.3 and 4.7.

4.3.3 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, 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 B. The results are presented and discussed in Sections 4.6.3 and 4.7.

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 eight features to Applet FSML, one feature to Struts FSML, and four features to WPI FSML. We removed one feature from WPI FSML.

4.3.4 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,

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

- 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 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:

$$A_f = C_f + M_f \tag{4.1}$$

$$Q_f = C_f + I_f \tag{4.2}$$

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

$$P_f = \frac{C_f}{Q_f} = \frac{C_f}{C_f + I_f} \tag{4.3}$$

$$R_f = \frac{C_f}{A_f} = \frac{C_f}{C_f + M_f} \tag{4.4}$$

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 4.5.

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 4.6.

4.4 Results of Phase 1: Code Patterns & Code Queries

Table 4.1: Mapping types for structural code patterns for Java (adapted from [13])

Structural Pattern Expression	**Structural Element(s) Matched**	**Abbreviation**
project	matches a Java project	project
p projectName	matches the name of the project *p*	projectName
class	matches a Java class	class
c fullyQualifiedName	matches the fully qualified name of the class *c*	fqName
c className	matches the simple name of the class *c*	className
c qualifier	matches the qualifier (package name) of the class *c*	qualifier
c assignableTo: *t* [concrete: r] [local: *p*]	matches if objects of the class *c* are assignable to the type *t*. The optional parameter *r* specifies that only concrete classes should be matched. The optional parameter *p* specifies that only source types of the project *p* should be matched	assignable
c isLocal: *p*	matches if the class *c* is a source class of the project *p*	isLocal
field	matches a field	field
f fieldName	matches the name of the field *f*	fieldName
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
x annotatedWith: *t*	matches a Java 5 annotation of type *t* placed on element *x*. The element *x* can be a class, a method, or a field	annotatedWith
a attribute: *n*	matches the value of attribute called *n* of a Java 5 annotation *a*	attribute
a hasNoAttribute	matches if the annotation *a* has no attributes (i.e., the annotation is a *marker* annotation)	noAttribute
mc argumentIsThis: *i* class: *c*	matches if the ith argument of the method call *mc* is a `this` literal assignable to the class *c*.	argIsThis
mc argumentIsField: *i* [sameAs: *f*]	matches if the ith argument of the method call *mc* is a field. Parameter *f* is optional and specifies a constraint that the matched field must be the same as the field that the feature `f` corresponds to.	argIsField
mc argumentIsNew: *i* signature: *s*	matches if the ith argument of the method call *mc* is a constructor call of the signature *s*	argIsNew
mc argumentIsVariable: *i*	matches if the ith argument of the method call *mc* is a variable	argIsVar

In order to retrieve framework-specific models, code patterns specified by the mapping definitions attached to features 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

Table 4.2: Mapping types for behavioural code patterns for Java (adapted from [13])

Behavioural Pattern Expression	Run-time Event Pattern(s) Matched	Abbrev.
c callsTo: s receiver: r [statement: s]	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. The optional parameter s specifies whether only method calls which are individual statements should be matched.	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 ith argument of the method call mc	argVal
	`argVal(): $mc && args(.., $i, ..)`	
c argument: i ofCall: mc_1 sameAsArg: j ofCall: mc_2	matches if the ith argument of the method call mc_1 points to the same object as the jth argument of the method call mc_2, in the control flow of objects of the class c	argSameObj
	`argSameObj($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 [subtypeOf: t]	matches assignments to the field f with an object returned by a constructor call with the signature cs. The optional parameter subtypeOf specifies that only constructor calls that create instance of the subtype of the type t should be matched	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);`	

resolved type and method bindings. Because run-time events do not exist statically, behavioural patterns consist of *shadows* [53] 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 Tables 4.1 and 4.2. The first column contains Smalltalk-like expressions that can be used to specify the patterns. The last column defines abbreviations used to refer to the given pattern type in the remainder of this paper. The second column presents descriptions of the semantics of code patterns. The description specifies the patterns in the code that match the given pattern expression. Since structural patterns can be fully 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 [65] 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 4.2, 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 [64] and some of its extensions, namely the Data Flow Pointcut [71] and Tracematches [5]. 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 4.2. 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`.

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

Figure 4.1: Fragment of the metamodel of the Applet FSML

```
[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>
```

Figure 4.2: Fragment of the metamodel of the Struts FSML

Figures 4.1-4.3 present fragments of the metamodels of the three FSMLs used in the study. 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.

We present the metamodels for two reasons: (i) to show fragments of the metamodels before they were modified in the elaboration phase, and (ii) to help the

```
[0..*] Part <class>
 ![1] implementsIView/IEditorPart <assignableTo: 'IViewPart/IEditorPart' concreteOnly: 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>
```

Figure 4.3: Fragment of the metamodel of the WPI FSML

reader understand the tables with precision and recall presented in Sections 4.5.1 and 4.6.3.

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.

We present the code queries that approximate behavioural patterns in Tables 4.3-4.10. Code queries are defined in the Smalltalk-like notation, similar to their corresponding pattern expressions in Tables 4.1 and 4.2. 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 analyzes that could infer any code property that could be statically inferred. These queries are discussed in Section 4.5. 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 implementation choices about the involved static analyzes, which could potentially reduce their precision and recall. The implementations are presented in Section 4.6.

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 4.3: Code queries for the *callsTo* mapping 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 4.4: Code queries for the *callsRec* mapping type

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

Table 4.5: Code queries for the *argVal* mapping type

Code Query Abbrev.	**Query Expression** / **Result**
argIsThis*	c thisAsArgument: i ofCall: mc_1 andArg: j ofCall: mc_2
	true iff both the ith argument of the method call mc_1 and the jth 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 ith argument of the method call mc_1 and the jth 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 4.6: Code queries for the *argSameObj* mapping type

Code queries presented in Tables 4.3-4.10 can be grouped based on the kind of approximation they employ. We discuss their potential false positives and false negatives.

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 4.7: Code queries for the *before* mapping 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 4.8: Code queries for the *retTypes* mapping 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 4.9: Code queries for the *assgnNew* mapping type

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

Table 4.10: Code queries for the *assignNull* mapping type

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

posed 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.

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

4.5.1 Precision & recall

Tables 4.11, 4.12, and 4.13 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 4.4. The last column, P_f, contains the precision calculated according to the equation 4.3. Except for three features which have a single false positive each, the precision is always 100% and we did not include it in the tables. In Section 4.5.2 we describe the refined queries and the data is discussed in Section 4.5.3.

4.5.2 The Refined Code Queries

In Tables 4.3-4.8, 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 4.11-4.13, that is, we measure false negatives with respect to the number of patterns matched by the best refined code query. In Section 4.3.4 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 `argIsPrv-FieldAO` 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.

4.5.3 Interpretation of the data

We discuss the data presented in Tables 4.11-4.13, each table separately. We focus on the highlighted cells.

Surprisingly, for all features except three, and for 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 4.4, 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 in a way that the queries did not return many false positives. In the following description we comment on the highlighted cells of Tables 4.11-4.13.

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 4.11: Statistics for framework-specific models retrieved using Applet FSML

Table 4.11. *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.

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 4.12: Statistics for framework-specific models retrieved using Struts FSML

Table 4.12. *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 4.13. *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.

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

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

The features `registersBeforeDeregisters` (of selection listeners). The 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`.

4.5.4 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 4.3-4.8, 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 [49]. 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.

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

4.6.1 Static analysis services used by the 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 4.2, building complete type hierarchy, call graph, and dataflow graph for framework-based applications is infeasible and, as shown in Section 4.5, 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.

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.

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.

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.

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.

4.6.2 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 4.5.3 when explaining the false negatives for the features `message` and `name` from Table 4.11, 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.

4.6.3 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 4.14-4.16.

Table 4.14 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 `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.

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 4.14: Statistics for framework-specific models retrieved using Applet FSML

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.

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 4.15: Statistics for framework-specific models retrieved using Struts FSML

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 4.15 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 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 4.16 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

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

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 over-

ridden (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 deregistration 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.

4.6.4 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 4.17 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 4.6.3. 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 4.17: Time and memory statistics for various analysis settings. The highlighted row contains values corresponding to the data presented in Section 4.6.3.

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 4.6.1, 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 `argIs-SameObject` 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 4.6.3, 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%.

4.7 Discussion

4.7.1 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 [68].

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 [35] and the built-in Eclipse Java Development Tools [40] 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 4.3.2). 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 4.3.3). 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.

4.7.2 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 [49]) and recent research on design fragments [46] 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.

4.7.3 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.

4.8 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 chapter 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. 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.

Chapter 5

Evaluation of Forward Engineering

5.1 Introduction

Forward engineering for framework-provided concepts is useful and needed. This argument is supported by the fact that many frameworks already provide configuration wizards that can be used by application developers to configure commonly used concepts and generate the initial implementation. In contrast to wizards, FSMLs usually offer richer variability by covering more concepts variants and implementation alternatives. Furthermore, FSMLs, as described in this thesis, also support incremental code addition, which amounts to being able to re-run the code generation with different parameter values on manually customized target code, which is usually not supported by wizards.

The possible benefits of code generation using FSMLs depend on the level of expertise of the developer. Developers new to the framework would like to know how the features can be implemented and they can learn framework completion from the default code generated for the features (by using the traceability links). They can also learn about the available implementation variants. Framework experts, on the other hand, are more interested in the automation aspect of code generation. They already know how to complete the framework and they are interested in automating tedious and repeatable tasks. Experts also take advantage of the incremental and global character of the code generation using FSMLs, since the implementation of the features can be inserted into existing code and it is usually scattered across the code.

In this chapter, we present a test we used to verify that the code generated for a correct model is the correct completion code with respect to the part of the API in scope of the used FSML. First, we describe code transformations we implemented for Java in Section 5.2. Next, we present the setup of the test in Section 5.3. We present and discuss the results in Section 5.4. Finally, we conclude the chapter in Section 5.5.

5.2 Code Transformations for Java

For the purpose of forward engineering we implemented code transformations for code pattern addition. The presented code transformations were designed to produce code patterns that will be matched by the corresponding code queries because the code transformations and the code queries implement two directions of the bidirectional mapping between the code and the framework-specific model. The bidirectional mapping is a function, that is, there exist many possible implementations of a single framework-specific model. For example, if a feature with the mapping definition <assignableTo: t> is present in a model that indicates that a certain class directly or indirectly implements/extends the type `t`. When generating the code for that same feature, certain decisions have to be made, such as, whether the type `t` or one of its subtypes should be used when adding an `extends` or `implements` declaration. Similarly, when generating method calls, a target method to the body of which the call is to be added has to be specified. In our implementation, certain decisions have been hard coded in the code transformations, for example, that the given type is extended/implemented directly for the `assignableTo` mapping type. Other decisions can be made by FSML designers by providing values to some additional parameters as compared to the mapping types they implement, for example, by specifying a certain method as a target method for call generation. Those additional parameters, called *forward parameters*, provide the detailed information needed for unambiguous code generation. Forward parameters are summarized in Table A.4.

Often, code transformations read information, such as class and field names, directly from the model. In these cases, transformation definition explicitly states that the values of certain parameters are retrieved from subfeatures with certain mapping types. For example, a class name can be retrieved from the subfeature with the mapping type `className` (cf. Table 5.1). Some transformations use the value of the feature they are executed for, for example, `addArgIsVar` (cf. Table 5.8).

Code Transformations for Structural Patterns

Tables 5.1-5.8 present code transformations for structural patterns from Table 4.1. Code transformation `addMethod` is also used for the mapping type `allMethods`.

<table>
<tr><th rowspan="2">Code Transformation Abbrev.</th><th>Transformation Expression | Default values for optional parameters</th></tr>
<tr><th>Result</th></tr>
<tr><td rowspan="2">addClass</td><td>p addClass: n [in: q] | q=”</td></tr>
<tr><td>creates a compilation unit with a class declaration named n in package q. Retrieves values of the parameters n and q from subfeatures with mapping types <code>className</code> and <code>qualifier</code> or <code>fullyQualifiedName</code></td></tr>
</table>

Table 5.1: Code transformation for the *class* mapping type

Code Transformation Abbrev.	Transformation Expression | Default values for optional parameters / Result
addAssignableTo	*c* addAssignableTo: *t* [concrete: *e*] | e=true
	If *t* is an interface, adds a `c implements t` superinterface declaration or adds *t* to the existing list of implemented interfaces. If *t* is a class, adds a `c extends t` superclass declaration. If *e*=true, adds implementations of the unimplemented methods of the superinterface or an abstract superclass

Table 5.2: Code transformation for the *assignableTo* mapping type

Code Transformation Abbrev.	Transformation Expression | Default values for optional parameters / Result
addField	*c* addField: *n* ofType: *t*
	adds a field declaration in the class *c* named *n* of type *t*. Retrieves values of the parameters *n* and *t* from subfeatures with mapping types `fieldName` and `fieldOfType`, respectively

Table 5.3: Code transformation for the *field* mapping type

Code Transformation Abbrev.	Transformation Expression | Default values for optional parameters / Result
addMethod	*c* addMethod: *s* [name: *n*]
	adds a method declaration of signature *s* in the class *c*. If method name *n* is given, replaces the name from the signature *s* with *n*. If the signature contains * for the method name, the parameter *n* is mandatory. Retrieves the value of the parameter *n* from a feature with the mapping type `methods`

Table 5.4: Code transformation for the *methods* mapping type

Code Transformation Abbrev.	Transformation Expression | Default values for optional parameters / Result
addArgIsThis	*mc* addArgIsThis: *i*
	adds the literal `this` as the ith argument of the method call *mc*

Table 5.5: Code transformation for the *argIsThis* mapping type

Code Transformation Abbrev.	Transformation Expression | Default values for optional parameters / Result
addArgIsField	*mc* addArgIsField: *i* [sameAs: *f*]
	adds the field *f* as the ith argument of the method call *mc*. If *f* is not specified, retrieves the name of the field from a subfeature with the mapping type `fieldName`

Table 5.6: Code transformation for the *argIsField* mapping type

Code Transformation Abbrev.	Transformation Expression | Default values for optional parameters / Result
addArgIsNew	*mc* addArgIsNew: *i* signature: *s*
	adds the constructor call of the signature *s* as the ith argument of the method call *mc*. Creates an anonymous subclass if necessary and adds implementation of unimplemented methods

Table 5.7: Code transformation for the *argIsNew* mapping type

Code Transformations for Behavioural Patterns

Tables 5.9-5.15 present code transformations for behavioural patterns from Table 4.2. There is no code transformation for the mapping type `before`. It is the responsibility of the FSML designer to specify target methods such that the first method call occurs before the second one.

Code Transformation Abbrev.	Transformation Expression | Default values for optional parameters / Result
addArgIsVar	mc addArgIsVar: i name:n signature: s
	adds a variable called n as the ith argument of the method call mc. Adds the variable declaration and initializes the variable with a constructor call of the signature s. Retrieves the value of the parameter n from the feature with the mapping type `argIsVar`

Table 5.8: Code transformation for the *argIsVar* mapping type

Code Transformation Abbrev.	Transformation Expression | Default values for optional parameters / Result
addCallTo	c addCallTo: s [receiverExpr: r] location: l [position: p] | r=", p=after
	creates a method call to a method with the signature s with the receiver expression r in the method of signature l of the class c at the position $p \in$ {before, after}

Table 5.9: Code transformation for the *callsTo* mapping type

Code Transformation Abbrev.	Transformation Expression | Default values for optional parameters / Result
addCallRec	c addCallTo: s [receiverExpr: r] location: l [position: p] | r=", p=after
	creates a method call to a method with the signature s with the receiver expression r in the method of signature l of the class c at the position $p \in$ {before, after}

Table 5.10: Code transformation for the *callsRec* mapping type

Code Transformation Abbrev.	Transformation Expression | Default values for optional parameters / Result
addArgVal	mc addArgVal: i [values: v]
	adds values of the ith argument of the method call mc. Adds a literal for a single value. For multiple values create a variable and multiple assignments with values v

Table 5.11: Code transformation for the *argVal* mapping type

Code Transformation Abbrev.	Transformation Expression | Default values for optional parameters / Result
addArgSameObj	c addArgument: i ofCall: mc_1 andArg: j ofCall: mc_2
	adds the literal `this` that resolves to class c as both the ith argument of the method call mc_1 and the jth argument of the method call mc_2

Table 5.12: Code transformation for the *argSameObj* mapping type

Code Transformation Abbrev.	Transformation Expression | Default values for optional parameters / Result
addRetTypes	m addReturnStms: c ifkey: i
	for each type t from the list of types c adds a return statement to the method m returning an object of that type. Adds each return statement at the beginning of the method and preceeds each return statement with an if statement of the form `if (t.class.equals(x)))`, where t is the type and x is the name of the method's i^{th} parameter.

Table 5.13: Code transformation for the *retTypes* mapping type

5.3 Setup of the Test

We evaluate forward engineering by verifying that the code generated for a correct framework-specific model is correct with respect to the fragment of the API in the scope of the used FSML. Any completion code can be easily checked for correctness by reverse engineering it and checking constraints on the model. Therefore, we

Code Transformation Abbrev.	Transformation Expression | Default values for optional parameters Result
addAssgnNew	*f* addAssignedNew[: *cc*] [initializer: *i*] [location: *l* position: *p*] [subtypeOf: *t*] | i=false, p=after
	Two variants: (1) adds an assignment to the field *f* with the constructor call of the signature *cc*. (2) adds an assignment to the field *f* with the constructor call to the default constructor of a subclass of the type *t*. Retrieves values of the parameters *n* and *q* from subfeatures with mapping types `className` and `qualifier` or `fullyQualifiedName`. Regardless of the variant: if *i*=true, adds the assignment in field's initializer; otherwise, adds the assignment in the method *l* at position $p \in$ {before, after}. Creates an anonymous subclass if necessary and adds implementation of unimplemented methods

Table 5.14: Code transformation for the *assgnNew* mapping type

Code Transformation Abbrev.	Transformation Expression | Default values for optional parameters Result
addAssgnNull	*f* getAssignedNull location: *l* [position: *p*] | p=after
	adds an assignment to the field *f* with the `null` literal in the method *l* at position *p* $\in$ {before, after}

Table 5.15: Code transformation for the *assgnNull* mapping type

can verify whether the forward engineered completion code is correct by reverse engineering it and comparing the new model with the original model used for forward engineering. If the original model was correct, the reverse engineered model should be correct as well and it should be *semantically equivalent* to the original one. Two identical models are semantically equivalent. Two different models are semantically equivalent if they contain exactly the same features modulo ordering and renaming. The original and the reverse engineered models may not be identical due to the ordering of the generated code patterns which may affect the order of features in the reverse engineered model.

Unfortunately, verifying the code generated for every possible correct model that can be expressed using a given FSML is infeasible since the number of all possible configurations of a feature model can be exponential to the number of features. Furthermore, if a feature model contains multiple features with unbounded cardinality (*), the number of possible configurations is infinite. Instead of verifying the code generated for every possible correct model, we manually build a model that has a good coverage of possible feature configurations. The latter is possible since many parts of the feature models of the exemplar FSMLs are independent and many values, such as class, field, and method names, can be considered as equivalent since they do not affect the conformance of the code to the framework API. Therefore, we created a single framework-specific model for each FSML that

- contains every feature of the given FSML at least once,
- covers all possible variants of implementing the concepts at least once in combinations that are significantly different,
- is correct with respect to the feature model.

The rationale behind such a model is that we want to avoid the combinatorial explosion when including all variants in all possible combinations and yet we want to

include all variants but only in important combinations. We choose the "important combinations" subjectively based on our knowledge of the language and we cannot present an exact algorithm. Instead, we present the complete models used in the test, the justification of their design, and the generated code, so that they can be judged objectively.

We executed the test for two FSMLs that support code generation for all of their features, WPI and Applet FSMLs, as follows. First, we created an empty Java project for Applet FSML and an empty plug-in project that can make UI contributions for WPI FSML. Next, we reverse engineered each project using the corresponding FSML. The resulting model for Applet FSML contained only the root feature that corresponds to the project (Figure 3.5, line $_1$). The resulting model for WPI FSML contained the root feature (Figure 3.3, line $_1$), 4 views, 1 editor, and 2 interactions that are implemented in the framework; we do not include them here because they do not affect the test in any way.

Next, we built the model as described above. We created 2 applets: `Applet1` (cf. Figure D.1) and `Applet2` (cf. Figure D.2), each with different kinds of listeners and threads. We created 3 views and 3 editors: `View1` (cf. Figure D.3), `View2` (cf. Figure D.4), `View3` (cf. Figure D.5), `Editor1` (cf. Figure D.6), `Editor2` (cf. Figure D.7), `Editor3` (cf. Figure D.8), each with different kinds of selection providers and selection and part listeners. Only `View3` and `Editor3` are adapter providers and only `View3` is an adapter requestor; the reason is to avoid unnecessary repetition. Also, we excluded one combination of features that is not excluded by the metamodel. The features `partListener` and `partListener2` cannot be added simultaneously because the argument of the two generated method calls (`this`) would have ambiguous type: `IPartListener` and `IPartListener2`, where the latter extends the former. In practice, only one type of the part listeners should be used at the same time. The reason why we do not forbid registering both types of part listener at the same time is because often a field is registered as a part listener of one type in example applications instead of the literal `this`. This variant with a field cannot be specified in the model using WPI FSML because the code transformation `addArgSameObj` does not offer an appropriate forward parameter. There is no such problem for selection listeners and `View1` registers all three kinds of listeners at the same time.

The left column of the Figures D.1-D.8 contains a framework-specific model created using the given FSML. The model is legal with respect to the FSML's metamodel. For brevity, we removed cardinality [1] of each feature instance, feature groups, and missing features. Grouped features retained their original indentation after the removal of feature groups, i.e., they are indented twice with respect to their parent features.

Finally, we generated the code and reverse engineered it to obtain the new model and we compared the new model with the original model used for code generation.

5.4 Results and Discussion

The right column of Figures D.1-D.8 presents the code generated using the FSMLs. We compacted the code by removing some empty lines, collapsing empty method bodies, and replacing implementations of interface methods and abstract superclass methods with "…".

For Applet FSML the reverse engineered model was identical with the original model. For WPI FSML, the new model was different from the original model in two ways. First, reverse engineering matched new behaviour inherited by `View3` from `PageBookView` and by `Editor3` from `MultiPageEditorPart` super classes: the class `PageBookView` registers a part listener and a selection provider; the class `MultiPageEditorPart` registers a selection provider. Second, the order of the features `adapters` (Figure 3.3, line 65) was reversed since the return statements are inserted at the beginning of the method body. The reverse engineered model was correct and equivalent to the original one except for the new behaviour inherited from the super classes.

In one case, the generated code contained a compile error: in `Editor3` the generated method `createPartControl` overrode the `final` method from the super class `MultiPageEditorPart`. In this case, the developer has to manually move statements from the body of the generated method to another method, for example, `addPages`. Supporting alternative target methods for all mapping types that can override methods remains future work.

Therefore, we can conclude that, with the exception described above, the code generated for a correct framework-specific model used in the test is the correct completion code with respect to the API covered by both FSMLs used in the test.

Code transformations encode two kinds of knowledge: programming language's syntax and semantics as well as programming micropatterns. In the first category, code transformations perform certain actions stipulated by the syntax and semantics of the programming language, such as, adding import declarations, adding methods required by super interfaces and abstract super classes, adding calls to super when overriding methods, adding return statements if necessary, including returning a value of the super method calls, adding super interface declarations when the literal `this` is used as an argument of method calls of the given type, and declaring a variable before its use.

The programming micropatterns implemented by the code transformations include creating anonymous sub types and implementing their methods, assigning fields in initializers, adding target methods for inserting method calls, and surrounding multiple return statements with appropriate `if` statements to prevent unreachability.

On top of these two generic kinds of knowledge, an FSML formalizes framework API knowledge by combining many features that are implemented by code transformations into higher-level abstractions. Framework-specific knowledge is

also used in setting the parameters of mapping definitions of features, such as, type names, method call signatures, and method call receiver's expressions. Forward engineering using FSMLs works because code transformations automate common implementation tasks that application developers perform when implementing instances of framework-provided concepts. Additionally, the structure of an FSML and the ordering of features enable FSML developer to control the order of code transformation executions and to ensure that certain code patterns are produced before others are, as needed. To summarize, the metamodel of an FSML can be seen as a *compositional* definition of a larger code transformation and the rules of metamodel interpretation and definitions of code transformations give the semantics of that transformation.

5.4.1 Threats to validity

In this section we discuss threats to internal and external validity of the evaluation.

Internal validity

The main threat to internal validity is the method of construction of the model used in the test, whereby, some combinations of features that could expose more compilation errors or incorrect API usage were not included.

To minimize this threat, but also avoid combinatorial explosion, we excluded only combinations of features that were redundant, that is, the generated code would be identical to the code already generated elsewhere. For example, `Applet1` has all listeners that register the literal `this` and all variants of threads that use a field. There is no need to include the same features in `Applet2`, instead we include all listeners that register a field, one more variant of a thread with a field and all variants of single task thread. Also, `Applet2` has two instances of the features `showStatus` and `parameter` with single values, whereas, `Applet1` has single instance of each of these features with two values each. It is not necessary to include three or more instances of these features nor is it necessary to include three or more values for each feature since the code will simply be repeated (this could be proven by induction). By the same reasoning, it is sufficient to include instances of the features `AdapterProvider` and `AdapterRequestor` only in `Editor3` and `View3`: including them elsewhere would simply duplicate the same code.

External validity

External validity is concerned with the generalization of the result, that is, how would this result generalize to other FSMLs.

In this test we only verified code transformations of mapping types related to Java that were used in the two FSMLs. Any other FSML with a correct metamodel

using the same mapping types should be capable of producing correct completion code under the condition that the code transformations satisfy the particular requirements of that FSML. As we have already shown in Chapter 3, mapping types, code queries, and code transformations constantly evolve to satisfy new requirements. We provided examples of such evolution. FSML engineering is an iterative process and the evaluation of forward engineering must be repeated for each language in the transition phase. However, in this chapter, we provided evidence that at least for two example languages forward engineering works and we designed a test that can be used for the evaluation of forward engineering for other FSMLs.

5.5 Conclusion

In this chapter we presented a test we performed to evaluate the correctness of the code generated using Applet and WPI FSMLs. We also described code transformations that were executed in the test. The results show that the code generated for different but valid combinations of features is correct with respect to the part of the API covered by an FSML. We discussed the results of the test as well as threats to its validity. The design of the test can be reused for the evaluation of forward engineering for other FSMLs.

Chapter 6

Evaluation of Round-Trip Engineering

6.1 Introduction

FSMLs can provide support for developers during the evolution of the completion code. The most basic support can be provided through the comparison of models reverse engineered at different times in the application's life cycle. The comparison of two models categorizes features into *unchanged*, *added*, *removed*, and *modified* as shown in Table 2.6. A feature is *modified* if its value has changed or any of its subfeatures were added, removed, or modified. The results of such comparison can be used for detecting both expected and unexpected changes of the completion code. For example, a designer may seek confirmation that their design expressed in a framework-specific model is still implemented in the current code and none of the features have been accidentally removed. A more advanced support can be provided through *round-trip engineering* in which both the model and the code can be independently modified and synchronized on demand using incremental updates. In round-trip engineering, conflicts can occur if incompatible changes are made in the model and in the code (cf. Table 2.6, rows 6, 7, 9, and 12); such conflicts can be automatically or manually resolved. Round-trip engineering can be useful if certain changes can be made more easily or more succinctly way in either the model or in the code and the developers can choose where to make the changes.

In this chapter we report on two tests we performed to evaluate round-trip engineering capability of two FSMLs: Applet and WPI. In particular, we verified that the code created incrementally in multiple iterations is equivalent to the code created during forward engineering and that the model created incrementally in multiple iterations is equivalent to the model created during reverse engineering. Because only a few code transformations used in the two FSMLs support code pattern modification and none of them support pattern removal, the test is limited to the addition of new features to the existing model and code. Supporting pattern modification and refactoring as well as safe pattern removal is very difficult and we

consider it future work. First, we present the setup of the test in Section 6.2. We present and discuss the results in Section 6.3. Finally, we conclude the chapter in Section 6.4.

6.2 Setup of the Tests

Each test involved multiple iterations of round-trip engineering as described in Figure 2.9 in Section 2.5.3. In both tests we used the same models and code we used in the evaluation of forward engineering. The first test verified whether the code created through multiple, incremental code pattern additions is equivalent to the code created in forward engineering. We performed the test as follows. First, we started with the model and an empty project as described in Section 5.3. Next, we performed the iterations. In each iteration, we first executed *reverse engineering* and *comparison* as shown in Figure 2.9. The result of the comparison, the *synchronization states*, describe that some features were added to the asserted model (cf. Table 2.6, row 10) and that some features were unchanged (cf. row 1). Next, we chose a feature that should be added to the code by making the reconciliation decision *enforce* for that feature; we chose features in the depth-first order. The chosen feature's parent should have had already been created in the code (its synchronization state should be *modified in A* (cf. row 5)). We also made the decision *enforce* for all of the chosen feature's essential and key subfeatures We left the mandatory and optional subfeatures (adding missing mandatory features in next iterations illustrates fixing incorrect code). For all remaining features we made the decision *ignore*. At the end of the iteration we executed *automatic reconciliation* which in turn executed code transformations for the features with the decision *enforce*. We iterated until all features had the synchronization state *consistent*. Finally, we compared the resulting code with the code generated during forward engineering.

The second test verified whether the model created through multiple, incremental updates is equivalent to the model created in reverse engineering. We performed the test as follows. First, we started with the generated code and empty model. Next, we performed the iterations. In each iteration, we first executed *reverse engineering* and *comparison* as shown in Figure 2.9. The result of the comparison, the *synchronization states*, describe that some features were added to the completion code (cf. Table 2.6, row 13) and that some features were unchanged (cf. row 1). Next, we chose a feature that should be added to the model by making the reconciliation decision *update* for that feature; we chose features in the depth-first order. The chosen feature's parent should have had already been added to the model (its synchronization state should be *modified in I* (cf. row 4)). We also made the decision *update* for all of the chosen feature's essential and key subfeatures. We left the mandatory and optional subfeatures (adding missing mandatory features in next iterations illustrates recognizing that incorrect code has been fixed). For all remaining features we made the decision *ignore*. At the end of the iteration

we executed *automatic reconciliation* which in turn executed model update for the features with the decision *update*. We iterated until all features had the synchronization state *consistent*. Finally, we compared the resulting model with the model created during reverse engineering.

The exact execution trace of both tests is presented in Appendix E. In each iteration we tried to reconcile as few features as possible (and maximize the number of iterations) to simulate a step by step evolution of the model and the code.

In the evaluation of forward engineering, we used a single model that contained important combinations of features in order to avoid combinatorial explosion related to huge number of possible models that can be expressed using an FSML. Round-trip engineering adds to that complexity: for every model or code, the features can be propagated in many different sequences. The order in which the features can be propagated is a *partial order* because parent features must always be propagated before subfeatures, essential and key subfeatures must always be propagated with their parent feature, and for some features in different places in the hierarchy the order does not matter. In this test, we choose a particular linear order of feature propagation out of all linear orders that satisfy the partial order imposed by the FSML's metamodel. We discuss this issue in threats to validity in Section 6.3.3.

6.3 Results and Discussion

6.3.1 Results of test 1

For Applet and WPI FSMLs the code created using round-trip engineering was equivalent with the code created using forward engineering, except for a few additional empty lines created during round-trip engineering. Below, we describe interesting issues encountered during the test for both FSMLs. All iteration numbers refer to line numbers of the execution trace in Appendix E.

Iteration 4. Iteration 23. Cannot add each instance of the features `message` and `name` individually. This is a limitation of our implementation in which a list of instances of these features in internally represented as a single instance with multiple values.

Iteration 70. Adding selection provider for `View3` requires manual conflict resolution because the super class `PageBookView` also registers a selection provider. We first enforce the feature `registers` from the model while ignoring the features from the code. Next, we update the model with the features implementing the inherited behaviour. At this point, the model and the code are fully consistent.

Iteration 73. The super class `PageBookView` already registers a part listener of this type and therefore the features have the synchronization state *added consistently* (cf. Table 2.6, row 3) and no reconciliation is necessary. Adding new code to `View3` for a part listener requires making a copy of the feature `registers` and

enforcing it, which we did to obtain the code similar to the one produced in forward engineering.

Iteration 74. After enforcing the features in this iteration and reverse engineering and comparison, we saw that the order of the features `adapter` was reversed. We reversed the features in the model to establish full consistency.

Iteration 94. The super class `MultiPageEditorPart` already registers a selection provider of this type and therefore the features have the synchronization state *added consistently* (cf. Table 2.6, row 3) and no reconciliation is necessary. Adding new code to `Editor3` for a selection provider requires making a copy of the feature `registers` and enforcing it, which we did to obtain the code similar to the one produced in forward engineering.

Iteration 98. The same as Iteration 74.

We also found out that we were unable to split the addition of part listeners (Iterations 60, 67, 73, 84, 91, and 97) into two iterations because the generated method call `addPartListener(null)` was ambiguous and could not be reverse engineered in the next iteration. The ambiguity results in a compile error because the compiler is unable to resolve method binding for the call (there are two possible target methods). To fix the problem, the argument of the method call should be cast either to `IPartListener` or `IPartListener2`, which requires adding a new forward parameter to the code transformations for the mapping types `callsTo` and `callsRec`.

6.3.2 Results of test 2

For Applet and WPI FSMLs the model created using round-trip engineering was equivalent with the model created using reverse engineering. It is important to note that we were able to update the model in more iterations as compared to the first test. For example, we could split iterations for part listeners into two iterations. The reason is that it is enough for the features added in a single iteration to be matched with that same features from the code in the next iteration, that is, the keys of the features added to the asserted model must be the same as the keys of the corresponding features in the implementation model. In other words, when selecting features for an iteration, their keys must not change. Otherwise, the features will not be considered as corresponding during comparison in the next iteration.

6.3.3 Threats to validity

In this section we discuss threats to internal and external validity of the evaluation.

Internal validity

The main threat to internal validity is that we did not perform feature addition in all possible orders that satisfy the partial order imposed by the FSML's metamodel. However, performing the test for all linear orders that satisfy the partial order is infeasible even when the test is fully automated since the number of all orders is extremely high (exponential).

In our FSMLs, however, many features are independent and they can be added in any order. Where the order matters, it has been made explicit by appropriate feature nesting and the ordering of essential features. Essential features must be all propagated together and they always are propagated according to their order specified in the FSML's metamodel. Sometimes the order is enforced by code transformations, for example, a variable is declared before it is used. For all other features the order does not matter. For example, it does not matter in which order the fields are created for threads or in which order listeners are registered. Note that, in general, in Java the order of fields actually matters but only if some fields are used in initializers of other field, they must be declared beforehand; we do not have such cases in our FSMLs.

Another threat to internal validity is related to the fact that we did not perform the tests on example applications, such that on those used in the evaluation of reverse engineering. In the tests for forward engineering, the result is predictable because the code is always created from scratch. Round-trip engineering must, on the other hand, work with any code, regardless if it was created manually or by code transformations executed previously.

However, any example application contains framework-completion code that conforms to the framework's API, for example, it implements framework-provided interfaces and overrides framework-provided methods. The mapping definitions of an FSML rely on the existence of those framework-provided patterns in the application code and code transformations create the patterns they require if they are missing. Therefore, we expect round-trip engineering to work the same way in real-life applications. We also assume, that the applications use the framework directly; we consider custom layers built on top of frameworks as new frameworks that require new FSMLs.

Possible problems we anticipate are name and type conflicts. For example, generating a field with an existing name would create a compilation error if the existing field had a different type (if it had the same type, the new field would not have been added because it would have matched a feature!). In some cases, code transformations generate unique identifiers, such as, in the case of the code transformation `addArgVal`, which declares a variable and multiple assignments to that variable. That transformation generates the name of the variable by appending consecutive integers to the name of the feature for which it is executed, if the same name already exists. A thorough study of possible conflicts that can occur in round-trip engineering of existing applications remains future work.

External validity

External validity is concerned with the generalization of the result, that is, how would this result generalize to other FSMLs.

Similarly to the evaluation of forward engineering, we verified only code transformations of mapping types related to Java that were used in the two FSMLs. Round-trip engineering should also work for other FSMLs using the same code transformations and with correct metamodels. In this chapter, we provided evidence that at least for two example languages round-trip engineering works and we designed a test that can be used for the evaluation of round-trip engineering for other FSMLs. Such a test should be performed in the transition phase of FSML development.

6.4 Conclusion

In this chapter we presented a test we performed to evaluate the correctness of the code and the model created using round-trip engineering. The test was limited to feature addition only. Both the code and the model were equivalent to the code created using forward engineering and the model created using reverse engineering, respectively. We performed the test for the same model and code as used in Chapter 6 and for a particular sequence of feature propagation. We discussed the threats to internal and external validity of such evaluation. Finally, the design of the test can be reused for the evaluation of round-trip engineering for other FSMLs.

Chapter 7

Evaluation of the Method

The presented method was extracted post mortem from our experience with engineering four exemplar FSMLs. Since the languages were not yet released to users, we cannot directly evaluate the degree in which the languages satisfy the use cases they were designed for and whether they achieve the value proposition. Such an evaluation would require significant effort. First, a direct evaluation would require gathering a user community and conducting exploratory studies by collecting users' feedback. Second, we focused our attention on the abstract syntax and semantics of the languages and we did not consider concrete syntax, which is an important factor in user studies since the quality of the user interface greatly influences usability of a modeling language and its adaptability by the end-users. We leave such an evaluation for future work. Instead, we evaluate the method and the exemplar languages, both quantitatively by using indirect measures and qualitatively by analysing threats to validity.

7.1 Evaluation of the Exemplar Languages

Since the method was designed based on the analysis of the exemplar languages, the quality of the languages impacts the quality of the method. In this section we evaluate to what degree the languages satisfy the use cases they were designed for.

7.1.1 Framework API understanding

Although only one of the exemplar languages (Applet FSML) was designed with this use case in mind, we think that FSMLs can be useful for learning framework concepts. The feature models are an explicit and concise summary of an aspect of the framework in scope of an FSML. Table 3.3 in Section 3.2 illustrates the way in which the information required to understand the FSML concepts is distributed across a variety of knowledge sources. The table shows that gaining understanding of the features requires referring to multiple sources and, therefore, it requires

significant effort on the part of the framework users. The users could learn the framework API by studying feature models and mapping definitions. Furthermore, augmenting features with information about the sources of knowledge from which the feature originated would allow the users to easily locate relevant fragments of documentation.

7.1.2 Completion code understanding and analysis

Based on the mapping definitions, a concrete *mapping* between features in a given feature configuration and patterns in given completion code can be established. Such a mapping, often referred to as *traceability links*, can be used by application developers to navigate from features to the implementation of the features and vice versa. The experimental evaluation of the quality of the retrieved models presented in Chapter 4 revealed 96% recall and 100% precision for behavioural code patterns. Since queries for structural patterns do not employ any approximations the precision and recall are 100% [13]. This result allows application developers to use the extracted framework-specific models with high confidence and easily locate code patterns using traceability links.

The model provides an alternative to code decomposition of the concept instance, in which all features are presented together in a single hierarchy. In contrast, code patterns corresponding to features are usually scattered across the application's code. The more the code patterns are scattered in the code, the greater the benefit of providing a model and traceability from the model to the code patterns.

In order to quantify the degree in which features from our exemplar FSMLs are scattered in the completion code, we measured the distribution of the matched code patterns over components and operations according to two metrics: *concern diffusion over components* (CDC) and *concern diffusion over operations* (CDO) [47]. In our case, the *components* are Java classes and XML documents, the *operations* are methods and XML elements, and the *concerns* are Java applets, Struts actions and forms, and Eclipse views and editors.

The metric CDC indicates the number of different components that the patterns implementing features of a concept instance (concern) are located in. Similarly, the metric CDO indicates the number of different operations that the patterns implementing features of a concept instance are located in. Note that some patterns, such as fields or superclass declarations, are not located in operations.

We extended the FSML infrastructure to automatically compute the values of these two metrics during reverse engineering. We reverse engineered a substantial number of concept instances: 76 Java applets, 158 Struts forms and actions, and 133 Eclipse views and editors. We list the applications used in Appendix C.

Table 7.1: Concern diffusion over components (CDC)

	Components	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**14**
WPI	Concerns	39	20	22	13	17	3	4	5	4	1	1	1	3
	Percentage	29.3%	15.0%	16.5%	9.8%	12.8%	2.3%	3.0%	3.8%	3.0%	0.8%	0.8%	0.8%	2.3%
Struts	Concerns	62	84	4	1	6	-	-	-	-	-	-	-	-
	Percentage	39.5%	53.5%	2.5%	0.6%	3.8%								
Applet	Concerns	66	10	-	-	-	-	-	-	-	-	-	-	-
	Percentage	89%	11%											

Table 7.2: Concern diffusion over operations (CDO)

	Operations	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**13**
WPI	Concerns	19	15	11	4	4	2	-	-	1	-	1
	Percentage	33.33%	26.32%	19.30%	7.02%	7.02%	3.51%			1.75%		1.75%
Struts	Concerns	53	5	25	8	9	9	4	3	1	2	-
	Percentage	44.5%	4.2%	21%	6.7%	7.6%	7.6%	3.4%	2.5%	0.8%	1.7%	
Applet	Concerns	19	15	11	4	4	2	-	-	1	-	1
	Percentage	33.3%	26.3%	19.3%	7%	7%	3.5%			1.8%		1.8%

Tables 7.1 and 7.2 present the distribution of concept instances over the given number of components and operations, respectively. Scattering over components depends much on the framework. 70% of Eclipse views and editors are implemented in two or more components and 55% of them in three or more components. The reasons include deep type hierarchies and the use of an XML configuration file.

In Struts, where only 7% of actions and forms are implemented in three or more components, type hierarchies are typically shallow but an XML configuration file is also used.

Java Applet is a very simple framework and hence applets are typically well modularized in one or two classes. Note that the Applet FSML does not support applet declarations on HTML pages, in which case applet features would be scattered over an additional component.

Looking at concern diffusion over operations, we can see that over 55% of concerns are implemented in two or more operations. The data supports our hypothesis that the code patterns implementing the features are typically scattered over the completion code and therefore the models provide an alternative to code decomposition of concept instances.

7.1.3 Completion code generation and evolution

In Chapters 5 and 6 we evaluated forward and round-trip engineering for Applet and WPI FSMLs. The evaluation confirmed the feasibility of forward and round-trip engineering using the two FSMLs and it presented the tests that can be used for the evaluation of other FSMLs. The evaluation of forward engineering showed that the completion code generated for a correct model is correct with respect to the API in scope of the FSML used in the test. The evaluation of round-trip engineering showed that the completion code created in multiple iterations is equivalent to the code generated in forward engineering and that the model created in multiple iterations is equivalent to the model created using reverse engineering.

FSMLs can also provide support when an API changes and the code has to be migrated to the new version of the API. In this case, an FSML can be extended with the new features and old features could be marked as *deprecated*. We added one such feature to the WPI FSML (Figure 3.3, line 22) as an example. After reverse engineering, the developers can see instances of deprecated features that need to be migrated. An automated refactoring could be devised to support such a migration.

Migration between two different, but conceptually similar, frameworks can also be supported. An example is semi-automated completion code migration from Apache Struts to Java Server Faces (JSF) framework [26]. In this example, the original code is first reverse engineered using the first version of the Struts FSML. Next, a specialized migration code generator is used to produce JSF compliant completion code. The code generator is guided by the model and it is capable of migrating not only the completion code but also the bodies of actions. The

automatic migration was demonstrated by migrating a real-world Struts-based application with 20 KLOC of Java to JSF.

7.2 Threats to Validity

We discuss the threats to external validity of the presented FSML engineering method. External validity is related to the extent in which the method can be applied to engineering new FSMLs for other frameworks.

The method is dependent on the approach to specification and implementation of FSMLs detailed in Chapter 2 rather than the particular generic infrastructure implementing it. Creating feature models, mapping definitions, mapping types, and code queries and transformations is an integral part of the method.

The applicability of the method is limited by the ability to model the concepts of frameworks using feature modeling and to specify the semantics of features using mapping definitions. As we indicated in Section 2.2, cardinality-based feature modeling with reference attributes and inheritance is a powerful concept modeling formalism whose expressiveness is comparable to class modeling and context-free grammars, which are typically used for modeling the abstract syntax of languages. Therefore cardinality-based feature modeling is appropriate for modeling framework concepts to the same degree as the other formalisms.

The semantics of features can be specified using the reusable set of mapping types we created and used in the exemplar languages. The set of mapping types defines a set of possible concepts that can be modeled because the concepts are inherently characterized by their features that correspond to code patterns. The set of mapping types we defined can be used to model a great variety of framework concepts, such as more static (structural) and more dynamic (behavioural) concepts as well as concepts spanning different artifact types (Java and XML). The current mapping types do not support concepts that are instantiated in the completion code by building complex object structures, such as graphical user interface (GUI) concepts. Implementation of code queries for matching such complex behavioural patterns requires sophisticated static analyses such as context- and flow-sensitive data flow analysis and points-to analysis. However, the method is not limited to this particular set of mapping types and new mapping types can be defined as needed. Also, the use of the context mechanism, which requires certain kind of feature nesting, for the retrieval of the values of dynamic parameters is voluntary: explicit paths can always be used.

Support for new programming languages, including both statically typed languages, such as C++ or C#, and dynamic languages, such as Ruby and JavaScript, can be added by defining new mapping types and implementing new mapping interpreters (that is code queries and transformations). Different language features, such as, multiple inheritance in C++, partial classes in C#, and automatic typing

in functional languages, require specialized mapping types and code queries implementing them. Code queries would require new static analysis services, such as, type inference for dynamically typed languages. Although we did not consider other languages than Java and XML, we think that existing work on static analyses can be leveraged for designing code queries for other languages and the study presented in Chapter 4 can be used for evaluating their effectiveness.

The degree to which FSMLs can achieve the value proposition and support the required use cases depends on the quality of the implementation of code queries and code transformations that implement mapping types used in the language. For example, a hypothetical mapping type could define a correspondence to the termination property of a program, determination of which is, in general, undecidable. Therefore, the possible support of the use cases is limited to the availability of code queries and transformations and the precision and recall of the approximations they may employ.

The ultimate confirmation of value proposition being delivered can be obtained from actual application developers. Although we have not performed any experiments or case studies involving real application developers, we have tested the ability of the FSMLs to support reverse engineering and migration on real-world applications. The forward engineering and round-trip engineering capabilities went through tests described in Chapters 5 and 6. Still, we believe that the impact of the feedback from FSML users on the method will be mostly related to refinement and fine-tuning of the individual activities and steps since all the main FSML definition artifacts, such as feature models; mapping types and definition; and code queries and transformations, need to be developed. User studies involving FSML application and FSML engineering are future work.

A potential threat is that the selection of frameworks used to create the exemplar FSMLs is not representative. The threat is addressed by the fact that the selected frameworks cover a range of basic framework properties relevant to FSML development. The selected frameworks cover up to two different artifact types. As argued above, supporting additional artifact types is not a limitation of this method. Another characteristic is whether a framework provides most of the functionality needed by an application or only a small part. The first kind of frameworks can be supported by full code generation, whereas the latter requires incremental code addition. Although we considered only frameworks in the latter category, full code generation is a special, simpler case of incremental code addition, so the method should be applicable to the first category, too. A major characteristic of a framework is whether it supports concept instantiation by subclassing and implementing interfaces or by instantiating objects and wiring them. All the considered frameworks fall in the first category. Handling the frameworks of the second category is likely to be challenging since less static structure will be available in the completion code. Applying the FSML approach to such frameworks is future work. Still the selected frameworks are widely used and representative of a large class of frameworks.

Framework-specific models expressed using the exemplar FSMLs are at a relatively low level of abstraction, which ensures certain stability needed for round-trip engineering. Defining mapping types that significantly raise the level of abstraction may limit the possibility of implementing effective and stable code queries and transformations, which, in turn, may limit support for certain use cases. We leave the method for engineering code queries and code transformations that support certain use cases for feature work.

Therefore, we think that the presented method for engineering FSMLs is general but limited to the expressiveness of feature models and the availability of mapping types and the code queries and transformations implementing them.

Because the method was created post-mortem we do not claim its completeness and usefulness; rather, the value of the method lies in the fact that it represents our experience and the results of the analysis of exemplar FSMLs.

The evaluation of the method's completeness and usefulness would require performing an experiment in which subjects without prior FSML engineering experience apply the method and evaluate its utility. We leave such evaluation for future work.

Chapter 8

Related Work

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

8.1 Framework Instantiation

Early approaches to supporting framework instantiation focused on providing API documentation in the form of a cookbook [70] that outlined development tasks and steps supplemented with code examples. Pree et al. showed how cookbooks can be made active and how development steps can automated [80]. Ortigosa et al. went even further by using intelligent agent technology that interactively guides the user in making subsequent implementation choices from the recipe [76].

In 1992, Johnson proposed documenting frameworks using patterns [60] that describe common usages of the framework. Such patterns describe the purpose and the design of a framework, and ways of using it.

Fontoura et al. proposed using a special profile of the UML language for representing framework's API called UML-F [48]. The profile defines annotations that can be placed on API elements to indicate their function, such as `variable` to indicate that the annotated method must be implemented or `extensible` to indicate that the annotated class may be extended. The users specialize a UML-F model to produce the design of their application and a supporting tool transforms the model into the code. The supporting tool is a wizard and it cannot be used for maintaining the consistency between the model and the code that is supported by round-trip engineering using FSMLs.

Hakala et al. proposed a pattern-based approach to generating task-driven programming environments for frameworks [51]. In this approach, specialization patterns are used to describe extension points of frameworks. Specialization patterns can be thought of as instances of more general design patterns. A specialized tool called FRamework EDitor (FRED) is used by application developers in the process of *casting*, that is, instantiating the given specialization pattern in the code.

FRED supports iterative and incremental casting which reflects the natural way programmers work.

Tourwe, in his Ph.D. thesis [92], proposed documenting framework's design using design patterns and then using such documentation to provide active support in framework-based software evolution. After manually identifying design pattern instances used in the framework, the tool can support framework evolution and instantiation related to these pattern instances. In particular, the tool can create subclasses of framework API classes and add empty method declarations in the application code to support framework instantiation. Application developers invoke code transformation for a given design pattern instance they wish to implement and the tool asks the developers to provide implementation for the created method bodies. The FSML approach is different by offering a conceptual view of the API rather than presenting the design of a framework as instances of design patterns. Similarly to the Tourwe's approach where the design pattern instances have to be manually identified, an FSML has to be manually built. Using FSMLs, the developers first configure a concept they wish to implement by selecting features and providing values of feature attributes and, later, they execute code transformations for the selected features in forward or round-trip engineering. The code transformations in FSMLs are not limited to creating classes and methods only but they can create any code patterns, including method calls, field assignments, and XML declarations. Tourwe argues that there is value in application developers knowing what design patterns were used in the design of the framework. However, the developers are not provided with the rationale for choosing to implement a particular pattern instance. In other words, for a complex framework, how do developers decide which pattern to implement? This problem is tackled in the FSML approach, whereby the developers are offered a choice of features that the application can implement and the configuration is guided by the feature and feature group cardinality constraints. Additional constraints allow verifying whether the feature configuration satisfies API constraints not captured by the hierarchy and cardinality constraints.

A newer approach proposed encoding common framework API engagement patterns as *design fragments* [46]. Such a design fragment has a name, a goal that can be accomplished by implementing it, a description of classes, methods, fields, and method and constructor calls, that need to be created to accomplish the goal, and a description of the relevant parts of the framework's API. Similarly to FSMLs, design fragments exist separately from the code. Developers choose a design fragment they wish to implement and manually bind it to their code. After the initial binding, the tool checks whether the required code elements were created and bound and reports violations. The developers continue creating code elements and binding them to the roles of a design fragment until all constraints are satisfied. In FSMLs, the binding of feature instances to the code patterns is established automatically during reverse engineering which discovers concept instances implemented in the code. After reverse engineering, the extracted framework-specific model can be checked for API constraint violations and missing features can be automatically added to the code using round-trip engineering. Furthermore, through variability

mechanisms, FSMLs can encode many alternative ways of implementing a single concept, each of which would require a separate design fragment.

Hou and Hoover presented an approach to checking structural constraints of applications using structural constraint language (SCL) [55]. SCL, previously known as framework constraint language (FCL) in Hou's Ph.D. thesis [56], can be used to encode API and other constraints and check them at compile time. The approach has been demonstrated on C++ and Java frameworks. In addition to typical program structure queries, SCL offers simple control-flow and data-flow-based constructs. Many of the API constraints present in the exemplar FSMLs could be written as SCL constraints.

8.2 Reverse Engineering

Chikofsky and Cross II defined reverse engineering as "the process of analyzing a subject system to a) identify the system's components and their interrelationships and b) create representations of the system in another form or at a higher level of abstraction" [27].

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 [36]. 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 interpreted 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 [84]. 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 [35], JTL [29], and CodeQuest [50] 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 4.2 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 4.2 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 [93] usually build a complete control flow graph of the application, which is a prerequisite for many other static analyses. However, as discussed in Section 4.2, 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 [83] 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. [96] 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 [77] 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.

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., [21]. 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 4.2. 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.

8.3 Forward Engineering

Chikofsky and Cross II defined forward engineering as "the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system" [27].

Template-based code generators. The most prevalent method of code generation in the industry today is *template-based code generation* in which the template is an incomplete target code with placeholders for the missing code. Complete target code is generated during the process of template expansion which is usually guided by a model. Template expansion fills the placeholders with the missing code based on the data from the model.

There are many examples of template-based code generator frameworks for different programming languages and many examples of using them for the generation of framework-based applications. Among most popular are code generators for Sun's Enterprise Java Beans framework, different Web Services frameworks, and many databases. References to many of them can be found on Code Generation Network [28].

However, the common problem with template-based code generation is supporting incremental code updates and preserving developer's customizations. Some template-based code generators, for example oAW [75], simply replace the entire target code during each generation. Other, such as JET [43], support different forms of *protected regions* and use merging to combine the old and the new code. A protected region is a place in the code designated by the user that should not be overridden by the code generator. Also, template-based code generators impose a very rigid structure on the generated code and only preserving the structure of the template enables subsequent code update during regeneration.

In contrast to the template-based approach, forward engineering in FSMLs is transformation-based and it has several advantages and disadvantages. The biggest advantage is that the transformation-based approach is incremental in nature: each code transformation is applied to the current code, regardless if the code was created by a previous code transformation or if it was written manually. In a compositional approach, such as FSMLs, larger code transformations are built by composing smaller ones. Transformation-based approach also supports integration of multiple code generators that modify the same code which is very difficult to achieve in template-based approach because of the rigid structure of the generated code. Also, code transformations in FSMLs implement common implementation steps and are reusable. Code templates, on the other hand, are usually much larger in scope and very rarely can be reused. Finally, incremental code transformations enable round-trip engineering (cf. Chapter 6) which is difficult to achieve in general when using code templates. The biggest disadvantage is that the implementation of code transformations is difficult and complete working applications most likely cannot be generated this way. We feel that a hybrid approach would work best in practice.

Authoring Refactorings. In our implementation, the code transformations are Java programs that manipulate the abstract syntax tree of a program using the API of Eclipse Java Development Tools [40]. An interesting approach to authoring code transformations is JunGL [94], a functional language for writing control-flow-based refactorings (e.g., extract method) for C#. Later, JunGL has been extended and applied for writing type-based refactorings (e.g., extract interface) [79]. One

of the great difficulties in building incremental code transformations is the ability to analyze the existing control and dataflow of the transformed program and seamlessly integrate the changes. JunGL provides such facilities in the form of *program path queries* which enable writing code transformations that are semantics preserving (refactorings). We think that JunGL would be very useful in implementing code transformations for FSMLs.

8.4 Round-Trip Engineering

Heterogeneous Synchronization. Round-trip engineering using FSMLs is a special case of the more general problem of heterogeneous synchronization, that is, propagation of changes among the artifacts of different types to establish or maintain a certain relation among the artifacts. We characterized round-trip engineering in FSMLs as *bidirectional*, *fully-incremental*, *many-to-one*, using *artifact translation*, *homogeneous artifact comparison and reconciliation with choice*, and *update translation with choice* [11]. The artifact translation operator translates the code into a framework-specific model (reverse engineering). The homogeneous artifact comparison and reconciliation with choice operator compares the asserted, implementation, and last-reconciled models (hence homogeneous). In FSMLs, the reconciliation is guided by the user who makes reconciliation decisions (hence the choice). The update translation with choice operator translates changes in the model to the changes of the code (code transformations). Finally, it is bidirectional and fully-incremental because both the model and the code are updated incrementally. It is important to note that the relation between the model and the code in FSMLs is a function because there are many different codes that have the same model. Therefore, the only place where the comparison and reconciliation is feasible is on the model side; comparing different versions of the code would be extremely difficult. We described many dimensions of heterogeneous synchronization and we presented sixteen concrete synchronizers and examples for some of them in the tutorial [11].

Evaluation of round-trip engineering. We are not aware of other works that evaluated round-trip engineering capabilities of other approaches. Many commercial UML modeling tools support round-trip engineering; however, we are not aware of any evidence of its usefulness and correctness.

8.5 Reengineering

Chikofsky and Cross II defined reengineering as "the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form" [27].

We can categorize different reengineering approaches as *generic* and *platform-specific*. Generic approaches are applicable to any software system implemented

in the supported programming language(es), for example, code refactorings that change the structure of the code while preserving its semantics [74]. Platform-specific approaches are only applicable to systems implemented on top of the supported platform. The second category is related to FSMLs and round-trip engineering can be seen as a form of reengineering. Another form of reengineering is migration to a different version of the same platform or to a different platform.

Reengineering of COBOL applications whose interactive part was implemented using Customer Information Control System (CICS) [85] is an example of generic reengineering. CICS is a transaction server and offers a specialized macro language for writing transactions which are then translated into the base language, e.g., COBOL, and then called by CICS in response to events (similarly to framework's callbacks). In [85] authors present a reengineering solution which removes the use of certain problematic constructs of CICS language that are translated into `go to` statements making the resulting program unstructured. The removal of such constructs improves the structure of the program without changing its behaviour.

Migration of WEB applications to new web platforms is an example of platform-specific reengineering. We already mentioned migration from Struts to Java Server Faces using FSMLs as an example [26]. Another example is *ajaxification*, that is, migration of multi-page web applications to Asynchronous JavaScript and XML model (AJAX) [73]. In AJAX model, a web application is displayed on a single page whose parts can be reloaded individually instead of reloading the entire page. To help with such a migration, authors propose first reverse engineering a multi-page web application into a *navigation model* and later clustering pages with similar URLs that could be included on a single page. Finally, the resulting *single page UI model* would be translated into an implementation for a particular AJAX framework.

8.6 FSML Engineering Method

Engineering domain-specific languages for APIs.

The idea of designing modeling languages for frameworks is not new. Roberts and Johnson proposed in 1996 that language tools, such as visual builders, should be the last stage of the evolution of a black-box framework [82]. They did not, however, present a systematic approach to building and implementing the languages embodied in such tools as the method presented in this thesis.

Bravenboer and Visser present MetaBorg [25], a method for embedding DSLs into general-purpose languages. A program expressed using an embedded DSL is included inside the program in a host language and, during the *assimilation* phase, it is translated into completion code for the API. In contrast, a model created using an FSML is external to the program in a host language. Code generation in MetaBorg is localized, that is, it replaces the embedded DSL program, whereas code generation in FSMLs has global and incremental character. Furthermore,

assimilation is performed during compilation and the developer is not intended to customize the generated code. In FSMLs, the generated code is integrated into the source code and it is intended to be customized.

Engineering domain-specific languages in general

In their 2005 survey [72], Mernik et al. distinguish the following domain-specific language (DSL) development phases: decision, analysis, design, implementation, and deployment. The survey includes the creation of a "user friendly" notation for an existing library or a framework as a possible motivation for the creation of a DSL in the decision phase. In the analysis phase, informal or formal domain analysis is performed to capture domain concepts by using implicit and explicit sources of knowledge, that include documentation, experience, and example code. Feature-oriented domain analysis (FODA) [61], which uses feature modeling, is included as a possible analysis method. The authors further state that no clear guidelines exist as to how a language could be designed from the results of the domain analysis. In our method, a basic feature-oriented domain analysis is refined to produce the entire definition and implementation of an FSML. Furthermore, our method gives specific guidelines how the concepts and features are obtained, scoped, structured, and mapped to code.

Kelly and Tolvanen present a process for designing domain-specific modeling languages (DSMLs) for 100% code generation [63]. They assume the existence of a domain framework and give general guidelines for building and adopting a DSML for that framework. They also assume that the framework is mature and complete such that the code generated for a model will never have to be manually modified. Furthermore, the modeling language concepts and elements of a DSML are requirements-level whereas the features of FSMLs are design-level.

Domain analysis techniques. Diaz and Arango [81] and Czarnecki and Eisenecker [30, ch. 2] surveyed numerous domain analysis techniques. Also, Czarnecki and Eisenecker give general guidelines for feature modeling [30, ch. 4] and compare different approaches to conceptual modeling [30, Apdx A]. These techniques and guidelines are, however, general-purpose whereas the method in this thesis is tailored for domain analysis of framework API concepts.

Attribute Grammars. Attribute grammars were first invented by Knuth to declaratively specify the computation of values of attributes contained by nodes of a tree [69]. The primary application of attribute grammars is specification of the computation of attribute values for nodes of an abstract syntax tree (AST) of a program. The computation of the values of attributes is specified using *attribute equations* and it can be carried out automatically after first determining the order of evaluation that respects dependencies between the equations.

FSMLs are similar to attribute grammars in that a metamodel of an FSML is also a declarative specification of computation over an abstract syntax tree of a model (i.e., feature configuration). The similarity is most visible in forward engineering, whereby the computation is the execution of code transformations for the nodes of the AST. Also, code patterns that feature instances correspond

to can be viewed as analogous to the values of attributes, that is, a treceability link to a code pattern can be thought of as a special traceability attribute of a feature. In forward engineering, code patterns are created for certain features (e.g., a class) and then used by code transformations for other features (e.g., adding a field to the previously created class). In reverse engineering, existing code patterns are matched in the code for certain features (e.g., a class) and then used by code queries for matching code patterns for other features (e.g., matching fields of a given type in the previously matched class).

The context mechanism used in FSMLs is similar to the *INCLUDING* construct from the LIGA system [62], which allows accessing an attribute from the closest ancestor node that has that attribute defined (context mechanism allows accessing a code pattern of the given type from the ancestor feature instance).

Currently, FSMLs employ a very primitive evaluation scheme, whereby the order of evaluation is predetermined by the hierarchy and ordering of features in the metamodel. Although, such a restriction imposes a certain structure of an FSML's metamodel, we did not find it too restrictive. We think, however, that FSMLs could benefit from the advanced attribute grammar evaluation approaches, such as remote attribute grammars [24], which allow remote access to the values of attributes of other nodes and provide algorithms for automatic scheduling of grammar evaluation.

Chapter 9

Conclusion

9.1 Summary of Contributions

In this thesis we presented a point of view that the existing support for application developers can be significantly improved by taking a language-oriented perspective on framework API. We presented framework-specific modeling languages (FSMLs) that formalize abstractions and rules of framework's application programming interfaces (APIs) and can express models of how applications use an API.

The language-oriented perspective means that API of a framework implicitly defines a domain-specific language. FSMLs formalize concepts of that language. Formalization is based on two assumptions: 1) API concepts can be modeled using feature models, and 2) an application consists of structural and behavioural code patterns that implement the features. We showed using numerous examples of concepts from four different frameworks that these were valid assumptions.

The view of an application as a composite code pattern was critical for being able to define different kinds of feature-to-code-pattern correspondences called mapping types. The set of mapping types and code queries and transformations implementing them is an important contribution as it enables round-trip engineering. We showed that the mapping types are generic (parameterized) and reusable.

Another important contribution is demonstration that a single language metamodel can be used for many different use cases. The metamodel can be interpreted manually in framework API understanding or automatically in reverse, forward, and round-trip engineering, as well as, in migration. We presented the metamodel interpretation algorithms.

We presented a method for engineering FSMLs that was extracted post mortem from the experience of building four such languages. The method is driven by the use cases that the FSMLs under development are to support. We presented the use cases, the overall process, and its instantiation for each language. The presentation focused on providing concrete examples for engineering steps, outcomes, and challenges. It also provides strategies for making engineering decisions. The

method represents a necessary step in the maturation of the FSML concept. We are not aware of any systematic approach to defining framework-provided concepts for the purpose of reverse, forward, and round-trip engineering other than the FSML approach presented in this thesis.

We evaluated the reverse engineering capability of three languages and the forward and round-trip engineering capabilities of two languages. The evaluation shows that these use cases can be performed using FSMLs. The tests we designed can be reused for the evaluation of these capabilities for other FSMLs.

Although, we did not directly evaluate the method and the exemplar languages in user studies, we provided some evidence of their usefulness using indirect measures. We showed that framework-specific models can be extracted from the application code with high recall and precision. We also showed that the models provide an alternative to code decomposition, that is, the features of a concept are presented together and the code patterns implementing them are typically scattered. We showed that the correct completion code can be created in forward engineering and we showed the correctness of round-trip engineering for two FSMLs.

The results presented in this thesis may impact the way framework APIs are constructed and documented. Taking a language-oriented perspective on framework APIs brings many benefits, such as formalization and automation. The need for FSMLs arises due to the inability of the programming language to adequately express API rules and constraints. Our hope is that programming languages will eventually evolve and become more suitable for API construction so that API rules will be checked by the compiler. However, until that happens FSMLs can provide support for the developers in the many ways described in this thesis and, hopefully, be an inspiration for programming language designers.

9.2 Limitations and Future Work

There are many possible directions for future work. Some of these were already mentioned in previous chapters; here we only mention directions for future work not included elsewhere.

One important area is the formalization of the FSML concept and the investigation if its properties. Such a formalization would offer deeper insights and would allow us to describe all components of FSMLs and the generic FSML infrastructure more precisely. In particular, three areas would benefit from formalization: FSML semantics, code queries and transformation, and round-trip engineering. Formalization of the semantic function and the semantic domain would provide justification for the design of the metamodel interpretation algorithms presented informally in Section 2.5 which, we think, implement such a function. Note that the notion of mapping type needs to be formalized as part of the semantics. Formalization of code queries and transformations is necessary to better understand the rules of their composition, such as, pre and post conditions. Code query and transformation

composition is important since the metamodel of an FSML can be understood as a compositional definition of a code query in reverse engineering and a code transformation in forward engineering. Formalization of round-trip engineering would provide insights to the properties of the approach, such as, what are exactly the conditions of reconciliation and what are the limitations of instance identification.

Application of FSMLs in practice is needed to ultimately confirm the usefulness and value of the concept. To that end, more languages should be built to address current problems with a few widely used frameworks, a user community should be established, and the effectiveness of FSMLs should be evaluated. Interesting studies can be performed by collecting and analyzing user survey data.

Another future work area related to the previous one is the creation of effective concrete syntax for FSMLs which is a prerequisite for any studies involving users. The creation of the concrete syntax should be driven by the kinds of concepts, for example, component-oriented or connector-oriented concepts require a *lines and boxes* type of interface or setting multiple related feature attributes requires a form interface. The concrete syntax should leverage results related to supporting feature model configuration, constraint propagation, and supporting different configuration interfaces, such as tree view and configuration wizards [7, 33, 34].

Application of the FSML engineering method in practice is needed to evaluate its usefulness and completeness. To that end, we envision conducting an experimental evaluation in which subjects apply the method to a well defined part of a framework's API, using provided documentation and example applications. Also, the FSML metamodeling tools would have to be developed to the degree that they are usable for the subjects. The tools should provide support for feature modeling, creating mapping definitions for mapping types implemented in the pluggable mapping interpreters, and development-time metamodel correctness checking (syntax and well typedness, cf. Section 2.5).

Through their support for round-trip engineering, many FSMLs can potentially be easily used in conjunction on the single code base. In general, language composition in still an open problem; however, we think FSMLs could be composed because the integration occurs in the completion code and each FSML covers a particular area of concern of the code. An example target scenario would involve developing a few FSMLs for the Eclipse Workbench framework, in addition to the WPI FSML, and evaluating possible interactions that can occur between the parts of the code related to different FSMLs.

APPENDICES

Appendix A

Mapping types, Constraints, and Forward parameters

Tables A.1-A.3 present descriptions of mapping types used in mapping definitions of features not related to Java. Mapping types for structural and behavioural patterns in Java were presented in Tables 4.1-4.2. These tables contain all mapping types used in all exemplars. Additionally, Table A.4 presents constraints and parameters used in forward engineering.

Table A.1: Mapping types for structural code patterns for plugin.xml

Structural Pattern Expression	**Structural Element(s) Matched**
p viewPartId: c	matches a value of the `partId` XML attribute of the view declaration corresponding to the class c in the `plugin.xml` file in the plug-in project p
p editorPartId: c	matches a value of the `partId` XML attribute of the editor declaration corresponding to the class c in the `plugin.xml` file in the plug-in project p
p contributor: c	matches a value of the `contributor` XML attribute of the editor declaration corresponding to the class c in the `plugin.xml` file in the plug-in project p

Table A.2: Mapping types for structural code patterns for XML

Structural Pattern Expression	**Structural Element(s) Matched**
p xmlDocument: h	matches a XML document at path h in project p
d xmlElement: n	matches a root XML element with the name n of the XML document d
e xmlElements: p	matches a XML elements at path p relative to XML element e
e xmlAttribute[: n]	matches a value of XML attribute called e of the XML element e. The parameter n is optional: the name of the feature is used in its absence
e xmlElementValue	matches a value of the XML element e

Table A.3: Constraints

Structural Pattern Expression	**Structural Element(s) Matched**
where: f contains: g	true if the value of the feature f contains the value of the feature g
where: f equalsTo: g	true if the value of the feature f is the same as the value of the feature g
valueEqualsTo: f value: v	true if the value of the feature f is the same as the value v
andParentIs: f	true if the the parent of the reference value is feature f
valueOf: f class: p	matches the value of the feature f which is the parent of feature p
constraint: f implies: g	false if the feature f is present and the feature g is missing. True otherwise (implication)

Table A.4: Constraints and parameters for forward engineering

Constraint	**Meaning**
subsumedBy: f	specifies that the code transformation for a feature should not be executed if the feature f is present
Parameter	**Meaning**
initializer: i	parameter i specifies whether a field assignment should be created in the field's initializer
location: s	specifies that the code should be inserted in the body of the method with the signature s
position: p	parameter p specifies whether the code should be inserted at the beginning (p=before) or at the end (p=after) of the method's body
receiverExpr: e	specifies the expression e that should be inserted as a receiver of a method call

Appendix B

Applications Used in the Third Phase of the Evaluation of Reverse Engineering

B.1 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.

B.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.

B.3 Applets

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

- 20 applets 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).

Appendix C

Applications Used in the Computation of the CDC and CDO Metrics

C.1 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.

C.2 Struts

The set applications used in the study consists of 3 applications: 2 example Struts applications shipped with the framework: Cookbook and Mailreader 1.3.8 and 1 large, open-source, production quality application: Apache Roller Weblogger 3.1.

C.3 Applets

The set of applets used in the study consists of 76 applets. The applets are divided in a two groups.

Sun. 20 applets examples shipped with the Java Development Kit.

Internet. 53 applet projects (56 applets) obtained from the internet by George Fairbanks and used in his study of design fragments [46]: 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.

Appendix D

Models Used in the Evaluation of Forward and Round-trip Engineering

In this Appendix we present the models and the code used in the tests presented in Chapters 5 and 6.

```
Applet
  name ('Applet1')
  extendsApplet
  overridesLifecycleMethods
      init
      start
      paint
      stop
      destroy
  showsStatus
    message ('status1')
    message ('status2')
  registersMouseListener
      this
        implementsMouseListener
        deregisters
          this
  registersMouseMotionListener
      this
        implementsMouseMotionListener
        deregisters
          this
  registersKeyListener
      this
        implementsKeyListener
        deregisters
          this
  Thread
    thread ('threadThis')
    typedThread
    InitializesThread
        initializesThreadWithRunnable
            this
              implementsRunnable
    nullifiesThread
  Thread
    thread ('threadField')
    typedThread
    InitializesThread
        initializesThreadWithRunnable
            runnableField
              typedRunnable
              name ('threadFieldRunnable')
              initialized
    nullifiesThread
  Thread
    thread ('threadHelper')
    typedThread
    InitializesThread
        initializesThreadWithRunnable
            helper
    nullifiesThread
  Thread
    thread ('threadVariable')
    typedThread
    InitializesThread
        initializesThreadWithRunnable
            variable ('runnableVariable')
    nullifiesThread
  parameter
    name ('param1')
    name ('param2')
  providesParameterInfo
  providesInfoForParameters
```

```
import java.applet.Applet;
import java.awt.Graphics;
import java.awt.event.MouseListener;
import java.awt.event.MouseMotionListener;
import java.awt.event.KeyListener;

public class Applet1 extends Applet implements MouseListener,
MouseMotionListener, KeyListener, Runnable {

  public Thread threadThis;
  public Thread threadField;
  public Runnable threadFieldRunnable = new Runnable() {
      ...
  };
  public Thread threadHelper;
  public Thread threadVariable;

  public void init() {
    super.init();
    String message0 = "status1";
    message0 = "status2";
    showStatus(message0);
    addMouseListener(this);
    addMouseMotionListener(this);
    addKeyListener(this);
    threadThis = new Thread(this);
    threadField = new Thread(threadFieldRunnable);
    threadHelper = new Thread(new Runnable() {
          ...
    });
    Runnable runnableVariable = new Runnable() {
          ...
    };
    threadVariable = new Thread(runnableVariable);
    String name0 = "param1";
    name0 = "param2";
    getParameter(name0);
  }

  public void start() {
    super.start();
  }

  public void paint(Graphics graphics) {}

  public void stop() {
    super.stop();
  }

  public void destroy() {
    super.destroy();
    removeMouseListener(this);
    removeMouseMotionListener(this);
    removeKeyListener(this);
    threadThis = null;
    threadField = null;
    threadHelper = null;
    threadVariable = null;
  }

  public String[][] getParameterInfo() {
    return super.getParameterInfo();
  }
}
```

Figure D.1: A model and the code generated using Applet FSML (`Applet1`)

```
Applet
  name ('Applet2')
  extendsApplet
    extendsJApplet
  overridesLifecycleMethods
      init
      destroy
  showsStatus
    message ('status1')
  showsStatus
    message ('status2')
  registersMouseListener
      mouseListenerField
        listenerField ('mListener')
        typedMouseListener
        initialized
        deregisters
          field
  registersMouseMotionListener
      mouseMotionListenerField
        listenerField ('mMotionListener')
        typedMouseMotionListener
        initialized
        deregisters
          field
  registersKeyListener
      keyListenerField
        listenerField ('keyListener')
        typedKeyListener
        initialized
        deregisters
          field
  Thread
    thread ('threadMyThread')
    typedThread
    InitializesThread
        initializesWithThreadSubclass
          name ('Applet2$MyThread')
          overridesRun
          extendsThread
    nullifiesThread
  singleTaskThread
      runnable
  singleTaskThread
      runnableField
        typedRunnable
        name ('singleTaskThreadRunnable')
        initialized
  parameter
    name ('param1')
  parameter
    name ('param2')
  providesParameterInfo
  providesInfoForParameters
```

```
import javax.swing.JApplet;
import java.awt.event.MouseListener;
import java.awt.event.MouseMotionListener;
import java.awt.event.KeyListener;

public class Applet2 extends JApplet {

  public MouseListener mouseListener = new MouseListener() {
    ...
  };
  public MouseMotionListener mouseMotionListener = new
MouseMotionListener() {
    ...
  };
  public KeyListener keyListener = new KeyListener() {
    ...
  };
  public Thread threadMyThread = new MyThread();
  public Runnable singleTaskThreadRunnable = new Runnable() {
    ...
  };

  public void init() {
    showStatus("status1");
    showStatus("status2");
    addMouseListener(mouseListener);
    addMouseMotionListener(mouseMotionListener);
    addKeyListener(keyListener);
    new Thread(new Runnable() {
      ...
    });
    new Thread(singleTaskThreadRunnable);
    getParameter("param1");
    getParameter("param2");
  }

  public void destroy() {
    removeMouseListener(mouseListener);
    removeMouseMotionListener(mouseMotionListener);
    removeKeyListener(keyListener);
    threadMyThread = null;
  }

  class MyThread extends Thread {
    public void run() {}
  }

  public String[][] getParameterInfo() {
    return null;
  }
  ...
}
```

Figure D.2: A model and the code generated using Applet FSML (Applet2)

```
ViewPart
  name ('View1')
  package ('parts')
  implementsIViewPart
SelectionProvider
  provider (View1)
  registers
      registersThis
        implementsISelectionProvider
SelectionListener
  listener (View1)
  registersAs
      globalSelectionListener
        deregisters
          deregistersSameObject
            registersBeforeDeregisters
      globalPostSelectionListener
        deregisters
          deregistersSameObject
            registersBeforeDeregisters
      specificSelectionListener
        registrationPartId ('View2.ID')
        deregisters
          deregistrationPartId ('View2.ID')
          deregistersSameObject
            registersBeforeDeregisters
PartListener
  listener (View1)
  registersAPartListener
      registers
        deregisters
          deregistersSameObject
            registersBeforeDeregisters
```

```
package parts;

import org.eclipse.ui.IViewPart;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.jface.viewers.ISelectionProvider;
import org.eclipse.ui.ISelectionListener;
import org.eclipse.ui.IPartListener;

public class View1 implements IViewPart, ISelectionProvider,
ISelectionListener, IPartListener {

  public void createPartControl(Composite composite) {
    getSite().setSelectionProvider(this);
    getSite().getPage().addSelectionListener(this);
    getSite().getPage().addPostSelectionListener(this);
    getSite().getPage().addSelectionListener("View2.ID", this);
    getSite().getPage().addPartListener(this);
  }

  public void dispose() {
    getSite().getPage().removeSelectionListener(this);
    getSite().getPage().removePostSelectionListener(this);
    getSite().getPage().removeSelectionListener("View2.ID",
this);
    getSite().getPage().removePartListener(this);
  }
  ...
}
```

Figure D.3: A model and the code generated using WPI FSML (`View1`)

```
ViewPart
  name ('View2')
  package ('parts')
  partId ('View2.ID')
  implementsIViewPart
    extendsViewPart
SelectionProvider
  provider (View2)
  registers
      registersHelper
SelectionListener
  listener (View2)
  registersAs
      globalPostSelectionListener
        deregisters
          deregistersSameObject
            registersBeforeDeregisters
PartListener
  listener (View2)
  registersAPartListener
      registers2
        deregisters
          deregistersSameObject
            registersBeforeDeregisters
```

```
package parts;

import org.eclipse.ui.part.ViewPart;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.jface.viewers.ISelectionProvider;
import org.eclipse.ui.ISelectionListener;
import org.eclipse.ui.IPartListener2;

public class View2 extends ViewPart implements
ISelectionListener, IPartListener2 {

  public void createPartControl(Composite composite) {
    getSite().setSelectionProvider(new ISelectionProvider() {
      ...
    });
    getSite().getPage().addPostSelectionListener(this);
    getSite().getPage().addPartListener(this);
  }
  public void dispose() {
    getSite().getPage().removePostSelectionListener(this);
    getSite().getPage().removePartListener(this);
  }
  ...
}
```

Figure D.4: A model and the code generated using WPI FSML (`View2`)

```
ViewPart
  name ('View3')
  package ('parts')
  implementsIViewPart
    extendsViewPart
      extendsPageBookView
SelectionProvider
  provider (View3)
  registers
      registersVariable ('selectionProviderVar')
SelectionListener
  listener (View3)
  registersAs
      specificSelectionListener
        registrationPartId ('View2.ID')
        deregisters
          deregistrationPartId ('View2.ID')
          deregistersSameObject
            registersBeforeDeregisters
PartListener
  listener (View3)
  registersAPartListener
      registers
        deregisters
          deregistersSameObject
            registersBeforeDeregisters
AdapterProvider
  provider (View3)
  providesAdapter
    adapters ('IContentProvider')
    adapters ('ILabelProvider')
AdapterRequestor
  requestor (View3)
  requestsAdapter
    adapter ('IContentProvider')
  requestsAdapter
    adapter ('ILabelProvider')
```

```
package parts;

import org.eclipse.ui.part.PageBookView;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.jface.viewers.ISelectionProvider;
import org.eclipse.ui.ISelectionListener;
import org.eclipse.ui.IPartListener;
import org.eclipse.jface.viewers.IContentProvider;
import org.eclipse.jface.viewers.ILabelProvider;
import org.eclipse.ui.IWorkbenchPart;

public class View3 extends PageBookView implements ISelectionListener, IPartListener {

  public void createPartControl(Composite composite) {
    super.createPartControl(composite);
    ISelectionProvider selectionProviderVar = new ISelectionProvider() {
      ...
    };
    getSite().setSelectionProvider(selectionProviderVar);
    getSite().getPage().addSelectionListener("View2.ID", this);
    getSite().getPage().addPartListener(this);
  }

  public void dispose() {
    super.dispose();
    getSite().getPage().removeSelectionListener("View2.ID", this);
    getSite().getPage().removePartListener(this);
  }

  public Object getAdapter(Class aClass) {
    if (ILabelProvider.class.equals(aClass))
      return new ILabelProvider() {
        ...
      };
    if (IContentProvider.class.equals(aClass))
      return new IContentProvider() {
        ...
      };
    return super.getAdapter(aClass);
  }

  public void requestAdapters(IWorkbenchPart iWorkbenchPart) {
    iWorkbenchPart.getAdapter(IContentProvider.class);
    iWorkbenchPart.getAdapter(ILabelProvider.class);
  }
  ...
}
```

Figure D.5: A model and the code generated using WPI FSML (`View3`)

```
EditorPart
  name ('Editor1')
  package ('parts')
  implementsIEditorPart
SelectionProvider
  provider (Editor1)
  registers
      registersField
        field ('selectionProviderFld')
        typedISelectionProvider
        initialized
SelectionListener
  listener (Editor1)
  registersAs
      globalSelectionListener
        deregisters
          deregistersSameObject
            registersBeforeDeregisters
PartListener
  listener (Editor1)
  registersAPartListener
      registers2
        deregisters
          deregistersSameObject
            registersBeforeDeregisters
```

```
package parts;

import org.eclipse.ui.IEditorPart;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.jface.viewers.ISelectionProvider;
import org.eclipse.ui.ISelectionListener;
import org.eclipse.ui.IPartListener2;

public class Editor1 implements IEditorPart, ISelectionListener,
IPartListener2 {

  public ISelectionProvider selectionProviderFld = new
ISelectionProvider() {
    ...
  };

  public void createPartControl(Composite composite) {
    getSite().setSelectionProvider(selectionProviderFld);
    getSite().getPage().addSelectionListener(this);
    getSite().getPage().addPartListener(this);
  }

  public void dispose() {
    getSite().getPage().removeSelectionListener(this);
    getSite().getPage().removePartListener(this);
  }
  ...
}
```

Figure D.6: A model and the code generated using WPI FSML (`Editor1`)

```
EditorPart
  name ('Editor2')
  package ('parts')
  implementsIEditorPart
    extendsEditorPart
SelectionProvider
  provider (Editor2)
  registers
      registersThis
        implementsISelectionProvider
SelectionListener
  listener (Editor2)
  registersAs
      globalPostSelectionListener
        deregisters
          deregistersSameObject
            registersBeforeDeregisters
PartListener
  listener (Editor2)
  registersAPartListener
      registers
        deregisters
          deregistersSameObject
            registersBeforeDeregisters
```

```
package parts;

import org.eclipse.ui.part.EditorPart;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.jface.viewers.ISelectionProvider;
import org.eclipse.ui.ISelectionListener;
import org.eclipse.ui.IPartListener;

public class Editor2 extends EditorPart implements
ISelectionProvider, ISelectionListener, IPartListener {

  public void createPartControl(Composite composite) {
    getSite().setSelectionProvider(this);
    getSite().getPage().addPostSelectionListener(this);
    getSite().getPage().addPartListener(this);
  }

  public void dispose() {
    getSite().getPage().removePostSelectionListener(this);
    getSite().getPage().removePartListener(this);
  }
  ...
}
```

Figure D.7: A model and the code generated using WPI FSML (`Editor2`)

```
EditorPart
  name ('Editor3')
  package ('parts')
  implementsIEditorPart
    extendsEditorPart
      extendsMultiPageEditorPart
SelectionProvider
  provider (Editor3)
  registers
      registersHelper
SelectionListener
  listener (Editor3)
  registersAs
      specificSelectionListener
        registrationPartId ('View2.ID')
        deregisters
          deregistrationPartId ('View2.ID')
          deregistersSameObject
            registersBeforeDeregisters
PartListener
  listener (Editor3)
  registersAPartListener
      registers2
        deregisters
          deregistersSameObject
            registersBeforeDeregisters
AdapterProvider
  provider (Editor3)
  providesAdapter
    adapters ('IContentProvider')
    adapters ('ILabelProvider')
```

```
package parts;

import org.eclipse.ui.part.MultiPageEditorPart;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.jface.viewers.ISelectionProvider;
import org.eclipse.ui.ISelectionListener;
import org.eclipse.ui.IPartListener2;
import org.eclipse.jface.viewers.IContentProvider;
import org.eclipse.jface.viewers.ILabelProvider;

public class Editor3 extends MultiPageEditorPart implements ISelectionListener, IPartListener2 {

  public void createPartControl(Composite composite) {
    super.createPartControl(composite);
    getSite().setSelectionProvider(new ISelectionProvider() {
      ...
    });
    getSite().getPage().addSelectionListener("View2.ID", this);
    getSite().getPage().addPartListener(this);
  }

  public void dispose() {
    super.dispose();
    getSite().getPage().removeSelectionListener("View2.ID", this);
    getSite().getPage().removePartListener(this);
  }

  public Object getAdapter(Class aClass) {
    if (ILabelProvider.class.equals(aClass))
      return new ILabelProvider() {
        ...
      };
    if (IContentProvider.class.equals(aClass))
      return new IContentProvider() {
        ...
      };
    return super.getAdapter(aClass);
  }
  ...
}
```

Figure D.8: A model and the code generated using WPI FSML (`Editor3`)

Appendix E

Traces of the Execution of the Tests in the Evaluation of Round-trip Engineering

In this Appendix we present the trace of execution of the tests presented in Chapter 6. Each line of the trace is a sequence of features surrounded in parentheses that were processed in a single iteration. We used line numbers to refer to the iterations.

```
1  (Applet, name ('Applet1'), extendsApplet),
2  (overridesLifecycleMethods, init),
3  (start, paint, stop, destroy),
4  (showsStatus, message ('status1'), message ('status2')),
5  (registersMouseListener, this, implementsMouseListener),
6  (deregisters, this),
7  (registersMouseMotionListener, this, implementsMouseMotionListener),
8  (deregisters, this),
9  (registersKeyListener, this, implementsKeyListener),
10 (deregisters, this),
11 (Thread, thread ('threadThis'), typedThread, InitializesThread, initializesThreadWithRunnable),
12 (this, implementsRunnable),
13 (nullifiesThread),
14 (Thread, thread ('threadField'), typedThread, InitializesThread,  initializesThreadWithRunnable),
15 (runnableField, typedRunnable, name ('threadFieldRunnable'), initialized),
16 (nullifiesThread),
17 (Thread, thread ('threadHelper'), typedThread, InitializesThread, initializesThreadWithRunnable),
18 (helper),
19 (nullifiesThread),
20 (Thread, thread ('threadVariable'), typedThread, InitializesThread, initializesThreadWithRunnable),
21 (variable ('runnableVariable')),
22 (nullifiesThread),
23 (parameter, name ('param1'), name ('param2')),
24 (providesParameterInfo),

26 (Applet, name ('Applet2'), extendsApplet, extendsJApplet),
27 (overridesLifecycleMethods, init, destroy),
28 (showsStatus, message ('status1')),
29 (showsStatus, message ('status2')),
30 (registersMouseListener, mouseListenerField, listenerField ('mListener'), typedMouseListener),
31 (initialized),
32 (deregisters, field),
33 (registersMouseMotionListener, mouseMotionListenerField, listenerField ('mMotionListener'),
   typedMouseMotionListener),
```

```
34 (initialized),
35 (deregisters, field),
36 (registersKeyListener, keyListenerField, listenerField ('keyListener'), typedKeyListener),
37 (initialized),
38 (deregisters, field),
39 (Thread, thread ('threadMyThread'), typedThread, InitializesThread, initializesWithThreadSubclass,
   name ('Applet2$MyThread'), extendsThread),
40 (overridesRun),
41 (nullifiesThread),
42 (singleTaskThread),
43 (runnable),
44 (singleTaskThread),
45 (runnableField, typedRunnable, name ('singleTaskThreadRunnable')),
46 (initialized),
47 (parameter, name ('param1')),
48 (parameter, name ('param2')),
49 (providesParameterInfo),

51 (ViewPart, name ('View1'), package ('parts'), implementsIViewPart),
52 (SelectionProvider, provider (View1), registers),
53 (registersThis, implementsISelectionProvider),
54 (SelectionListener, listener (View1), registersAs, globalSelectionListener),
55 (deregisters, deregistersSameObject),
56 (globalPostSelectionListener),
57 (deregisters, deregistersSameObject),
58 (specificSelectionListener, registrationPartId ('View2.ID')),
59 (deregisters, deregistrationPartId ('View2.ID'), deregistersSameObject),
60 (PartListener, listener (View1), registersAPartListener, registers, deregisters,
   deregistersSameObject),
62 (ViewPart, name ('View2'), package ('parts'), partId ('View2.ID'), implementsIViewPart,
   extendsViewPart),
63 (SelectionProvider, provider (View2), registers),
64 (registersHelper),
65 (SelectionListener, listener (View2), registersAs, globalPostSelectionListener),
66 (deregisters, deregistersSameObject),
67 (PartListener, listener (View2), registersAPartListener, registers2, deregisters,
   deregistersSameObject),
69 (ViewPart, name ('View3'), package ('parts'), implementsIViewPart, extendsViewPart,
   extendsPageBookView),
70 (SelectionProvider, provider (View3), registers, registersVariable ('selectionProviderVar')),
71 (SelectionListener, listener (View3), registersAs, specificSelectionListener, registrationPartId
   ('View2.ID')),
72 (deregisters, deregistrationPartId ('View2.ID'), deregistersSameObject),
73 (PartListener, listener (View3), registersAPartListener, registers, deregisters,
   deregistersSameObject),
74 (AdapterProvider, provider (View3), providesAdapter, adapters ('IContentProvider'), adapters
   ('ILabelProvider')),
75 (AdapterRequestor, requestor (View3), requestsAdapter, adapter ('IContentProvider'),
76 (requestsAdapter, adapter ('ILabelProvider')),

78 (EditorPart, name ('Editor1'), package ('parts'), implementsIEditorPart),
79 (SelectionProvider, provider (Editor1), registers),
80 (registersField, field ('selectionProviderFld'), typedISelectionProvider),
81 (initialized),
82 (SelectionListener, listener (Editor1), registersAs, globalSelectionListener),
83 (deregisters, deregistersSameObject),
84 (PartListener, listener (Editor1), registersAPartListener, registers2, deregisters,
   deregistersSameObject),
86 (EditorPart, name ('Editor2'), package ('parts'), implementsIEditorPart, extendsEditorPart),
87 (SelectionProvider, provider (Editor2), registers),
88 (registersThis, implementsISelectionProvider),
89 (SelectionListener, listener (Editor2), registersAs, globalPostSelectionListener),
90 (deregisters, deregistersSameObject),
91 (PartListener, listener (Editor2), registersAPartListener, registers, deregisters,
   deregistersSameObject),
93 (EditorPart, name ('Editor3'), package ('parts'), implementsIEditorPart, extendsEditorPart,
   extendsMultiPageEditorPart),
94 (SelectionProvider, provider (Editor3), registers, registersHelper),
```

```
95  (SelectionListener, listener (Editor3), registersAs, specificSelectionListener, registrationPartId
    ('View2.ID')),
96  (deregisters, deregistrationPartId ('View2.ID'), deregistersSameObject)),
97  (PartListener, listener (Editor3), registersAPartListener, registers2, deregisters,
    deregistersSameObject),
98  (AdapterProvider, provider (Editor3), providesAdapter, adapters ('IContentProvider'), adapters
    ('ILabelProvider'))
```

Appendix F

Complete Metamodels of the Exemplar FSMLs

We present full metamodels of the exemplar FSMLs in the following sections. We removed some information related to the implementation the metamodels and transformed them for better presentation. The metamodels retain the key information. We removed the qualifiers of all types, also those used in method signatures. For example, `java.lang.String` became `String`. The mapping definitions are presented according to mapping types from Tables 4.1-4.2 and A.1-A.3. The metamodels also contain some additional parameters needed for code generation described in Table A.4.

F.1 Metamodel of WPI FSML

```
1  [1..1] WorkbenchPartInteractions <project>
2    [1..1] project <projectName>
3    [0..*] Part <class>
4     ![1..1] name (String) <className>
5      [0..1] package (String) <qualifier>
6      [0..1] local <isLocal>
7    [0..*] ViewPart -|> Part
8      [0..1] partId (String) <viewPartId>
9     ![1..1] implementsIViewPart <assignableTo: 'IViewPart' concrete: true> <subsumedBy: extendsViewPart/ >
10       [0..1] extendsViewPart <assignableTo: 'ViewPart'> <subsumedBy: extendsPageBookView>
11         [0..1] extendsPageBookView <assignableTo: 'PageBookView'>
12   [0..*] EditorPart -|> Part
13     [0..1] partId (String) <editorPartId>
14     [0..1] contributor <class>
15      ![1..1] contributor (String) <editorContributor> <fullyQualifiedName>
16       [1..1] extendsEditorActionBarContributor <assignableTo: 'EditorActionBarContributor'> <subsumedBy: extendsMultiPageEditorActionBarContributor>
17         [0..1] extendsMultiPageEditorActionBarContributor <assignableTo: 'MultiPageEditorActionBarContributor'>
18       [1..1] multiPageEditorAndContributor <constraint ../../extendsEditorPart/extendsMultiPageEditorPart implies: ../extendsEditorActionBarContri-
butor/extendsMultiPageEditorActionBarContributor>
19    ![1..1] implementsIEditorPart <assignableTo: 'IEditorPart' concrete: true> <subsumedBy: extendsEditorPart/>
20       [0..1] extendsEditorPart <assignableTo: 'EditorPart'> <subsumedBy: extendsMultiPageEditorPart>
21         [0..1] extendsMultiPageEditorPart <assignableTo: 'MultiPageEditorPart'>
22       [0..0] extendsMultiPageEditor <assignableTo: 'MultiPageEditor'> <deprecated>
23   [0..*] SelectionProvider <class>
24    ![1..1] provider (Part) <baseConcept>
25    ![1..*] registers <callsTo: 'void IWorkbenchSite.setSelectionProvider(ISelectionProvider)' receiverExpr: 'getSite()' location: 'void createPart-
Control(Composite)' position: 'after'>
26       <1-1>
27         [0..1] registersThis <argumentIsThis: 1>
28           [1..1] implementsISelectionProvider <assignableTo: 'ISelectionProvider'>
29         [0..1] registersHelper <argumentIsNew: 1 signature: 'void ISelectionProvider()'>
30         [0..1] registersVariable <argumentIsVariable: 1 signature: 'void ISelectionProvider()'>
31         [0..1] registersField <argumentIsField: 1>
32           [1..1] field (String) <fieldName>
33           [1..1] typedISelectionProvider <typedWith: 'ISelectionProvider'>
34           [1..1] initialized <assignedWithNew: 'void ISelectionProvider()' initializer: true>
35   [0..*] SelectionListener <class>
36    ![1..1] listener (Part) <baseConcept>
37    ![1..1] registersAs
38      !<1-3>
39         [0..*] globalSelectionListener <callsTo: 'void ISelectionService.addSelectionListener(ISelectionListener)' receiverExpr: 'getSite().-
getPage()' position: 'after' location: 'void createPartControl(Composite)'>
```

```
40          [1..*] deregisters <callsTo: 'void ISelectionService.removeSelectionListener(ISelectionListener)' receiverExpr: 'getSite().getPage()' location: 'void dispose()' position: 'after'>
41           ![1..1] deregistersSameObject <argument: 1 ofCall: ../ sameAsArg: 1 ofMethodCall: ../../>
42              [1..1] registersBeforeDeregisters <methodCall call: ../../../ before: ../../ givenCallbackSeq: 'init createPartControl dispose'>
43        [0..*] globalPostSelectionListener <callsTo: 'void ISelectionService.addPostSelectionListener(ISelectionListener)' receiverExpr: 'getSite().getPage()' position: 'after' location: 'void createPartControl(Composite)'>
44          [1..*] deregisters <callsTo: 'void ISelectionService.removePostSelectionListener(ISelectionListener)' receiverExpr: 'getSite().getPage()' location: 'void dispose()' position: 'after'>
45           ![1..1] deregistersSameObject <argument: 1 ofCall: ../ sameAsArg: 1 ofMethodCall: ../../>
46              [1..1] registersBeforeDeregisters <methodCall call: ../../../ before: ../../ givenCallbackSeq: 'init createPartControl dispose'>
47        [0..*] specificSelectionListener <callsTo: 'void ISelectionService.addSelectionListener(String, ISelectionListener)' receiverExpr: 'getSite().getPage()' position: 'after' location: 'void createPartControl(Composite)'>
48         ![1..1] registrationPartId (String) <valueOfArg: 1>
49          [1..1] provider (Part) <where attribute: partId equalsTo: ../registrationPartId>
50          [0..1] providerName
51          [1..*] deregisters <callsTo: 'void ISelectionService.removeSelectionListener(String, ISelectionListener)' receiverExpr: 'getSite().getPage()' location: 'void dispose()' position: 'after'>
52           ![1..1] deregistrationPartId (String) <valueOfArg: 1> <valueEqualsTo attribute: ../../registrationPartId>
53           ![1..1] deregistersSameObject <argument: 2 ofCall: ../ sameAsArg: 2 ofMethodCall: ../../>
54              [1..1] registersBeforeDeregisters <methodCall call: ../../../ before: ../../ givenCallbackSeq: 'init createPartControl dispose'>
55  [0..*] PartListener <class>
56   ![1..1] listener (Part) <baseConcept>
57   ![1..1] registersAPartListener
58     !<1-2>
59        [0..*] registers <callsTo: 'void IPartService.addPartListener(IPartListener)' position: 'after' receiverExpr: 'getSite().getPage()' location: 'void createPartControl(Composite)'>
60          [1..*] deregisters <callsTo: 'void IPartService.removePartListener(IPartListener)' receiverExpr: 'getSite().getPage()' location: 'void dispose()' position: 'after'>
61           ![1..1] deregistersSameObject <argument: 1 ofCall: ../ sameAsArg: 1 ofMethodCall: ../../>
62              [1..1] registersBeforeDeregisters <methodCall call: ../../../ before: ../../ givenCallbackSeq: 'init createPartControl dispose'>
63        [0..*] registers2 <callsTo: 'void IPartService.addPartListener(IPartListener2)' position: 'after' receiverExpr: 'getSite().getPage()' location: 'void createPartControl(Composite)'>
64          [1..*] deregisters <callsTo: 'void IPartService.removePartListener(IPartListener2)' receiverExpr: 'getSite().getPage()' location: 'void dispose()' position: 'after'>
65           ![1..1] deregistersSameObject <argument: 1 ofCall: ../ sameAsArg: 1 ofMethodCall: ../../>
66              [1..1] registersBeforeDeregisters <methodCall call: ../../../ before: ../../ givenCallbackSeq: 'init createPartControl dispose'>
67  [0..*] AdapterProvider <class>
68   ![1..1] provider (Part) <baseConcept>
69   ![1..1] providesAdapter <allMethods: 'Object getAdapter(Class)'>
70     ![1..*] adapters (String) <returnedObjectTypes ifkey: 1>
71  [0..*] AdapterRequestor <class>
72   ![1..1] requestor (Part) <baseConcept>
73   ![1..*] requestsAdapter <callsTo: 'Object IAdaptable.getAdapter(Class)' receiver: 'IWorkbenchPart' position: 'after' location: 'void requestAdapters(IWorkbenchPart)' receiverExpr: 'iWorkbenchPart'>
74      [1..1] adapter (String) <valueOfArg: 1>
75      [0..*] adapterProvider (AdapterProvider) <where attribute: providesAdapter/adapters contains: ../adapter>
```

F.2 Metamodel of Struts FSML

```
[1..1] StrutsApplication <project>
 ![1..1] name (String) <projectName>
  [1..1] StrutsConfig <xmlDocument: '/WEB-INF/struts-config.xml'> <xmlElement name: 'struts-config'>
    [0..*] FormDecl <xmlElements: 'form-beans/form-bean'> <xmlElement>
      [1..1] name (String) <xmlAttribute>
      [1..1] formType (String) <xmlAttribute: 'type'>
      [0..1] isDynaActionForm <valueEqualsTo attribute: ../formType value: 'DynaActionForm'>
      [0..*] formProperty <xmlElements: 'form-property'> <xmlElement>
        [1..1] name (String) <xmlAttribute>
        [0..1] type (String) <xmlAttribute>
    [0..*] ForwardDecl <xmlElements: 'global-forwards/forward'> <xmlElement>
      [1..1] name (String) <xmlAttribute>
      [0..1] path (String) <xmlAttribute>
      [1..1] target (ActionDecl) <where attribute: path equalsTo: ../path>
    [0..*] ActionDecl <xmlElements: 'action-mappings/action'> <xmlElement>
      [1..1] path (String) <xmlAttribute>
      [0..1] name (String) <xmlAttribute>
      [0..1] type (String) <xmlAttribute>
      [1..1] actionImpl (ActionImpl) <where attribute: qualifiedName equalsTo: ../type>
      [0..*] forwards <xmlElements: 'forward'> <xmlElement>
        [1..1] name (String) <xmlAttribute>
        [0..1] path (String) <xmlAttribute>
        [1..1] target (ActionDecl) <where attribute: path equalsTo: ../path>
      [0..1] input (String) <xmlAttribute>
  [0..*] FormImpl <class>
   ![1..1] name (String) <className>
    [0..1] package (String) <qualifier>
    [1..1] qualifiedName (String)
    [0..1] local <isLocal>
   ![1..1] extendsActionForm <assignableTo: 'ActionForm'> <subsumedBy: extendsDynaActionForm>
      [0..1] extendsDynaActionForm <assignableTo: 'DynaActionForm'>
  [0..*] ActionImpl <class>
   ![1..1] name (String) <className>
    [0..1] package (String) <qualifier>
    [1..1] qualifiedName (String)
    [0..1] local <isLocal>
   ![1..1] extendsAction <assignableTo: 'Action'> <subsumedBy: extendsDispatchAction>
      [0..1] extendsDispatchAction <assignableTo: 'DispatchAction'>
        [0..*] actionMethod (String) <methods: 'ActionForward *(ActionMapping, ActionForm, HttpServletRequest, HttpServletResponse)'>
      [0..1] overridesExecute <methods: 'ActionForward execute(ActionMapping, ActionForm, HttpServletRequest, HttpServletResponse)'>
    [0..*] forwards <callsTo: 'ActionForward ActionMapping.findForward(String)'>
```

```
42        [1..1] name (String) <valueOfArg: 1>
43        [1..1] forward
44          <1-2>
45            [0..1] localForward (ForwardDecl) <where attribute: name equalsTo: ../../name> <and attribute: ../type equalsTo: ../../../qualifiedName>
46            [0..1] globalForward (ForwardDecl) <where attribute: name equalsTo: ../../name> <andParentIs instanceOf: 'StrutsConfig'>
47      [0..*] inputForwards <callsTo: 'ActionForward ActionMapping.getInputForward()'>
48        [1..1] name (String) <valueOf attribute: input class: 'ActionDecl'> <where attribute: type equalsTo: ../../qualifiedName>
```

F.3 Metamodel of Applet FSML

```
1   AppletModel <project>
2     [0..*] Applet <class>
3       [1..1] name (String) <fullyQualifiedName>
4      ![1..1] extendsApplet <assignableTo: 'Applet' local: true> <subsumedBy: extendsJApplet>
5         [0..1] extendsJApplet <assignableTo: 'JApplet'>
6       [0..1] overridesLifecycleMethods
7        !<1-5>
8           [0..1] init <methods: 'void init()'>
9           [0..1] start <methods: 'void start()'>
10          [0..1] paint <methods: 'void paint(Graphics)'>
11          [0..1] stop <methods: 'void stop()'>
12          [0..1] destroy <methods: 'void destroy()'>
13      [0..*] showsStatus <callsReceived: 'void Applet.showStatus(String)' location: 'void init()' position: 'after'>
14        [0..*] message (String) <valueOfArg: 1>
15      [0..*] registersMouseListener <callsTo: 'void Component.addMouseListener(MouseListener)' position: 'after' location: 'void init()'>
16       !<1-1>
17          [0..1] this <argumentIsThis: 1>
18            [1..1] implementsMouseListener <assignableTo: 'MouseListener'>
19            [1..1] deregisters <callsTo: 'void Component.removeMouseListener(MouseListener)' location: 'void destroy()'>
20             ![1..1] this <argumentIsThis: 1>
21          [0..1] mouseListenerField <argumentIsField: 1> <field>
22            [1..1] listenerField (String) <fieldName>
23            [1..1] typedMouseListener <fieldOfType: 'MouseListener'>
24            [1..1] initialized <assignedNew: 'void MouseListener()' initializer: true>
25            [1..1] deregisters <callsTo: 'void Component.removeMouseListener(MouseListener)' location: 'void destroy()'>
26             ![1..1] field <argumentIsField: 1 sameAs: ../../listenerField>
27      [0..*] registersMouseMotionListener <callsTo: 'void Component.addMouseMotionListener(MouseMotionListener)' position: 'after' location: 'void
    init()'>
28       !<1-1>
29          [0..1] this <argumentIsThis: 1>
30            [1..1] implementsMouseMotionListener <assignableTo: 'MouseMotionListener'>
31            [1..1] deregisters <callsTo: 'void Component.removeMouseMotionListener(MouseMotionListener)' location: 'void destroy()'>
```

```
32           ![1..1] this <argumentIsThis: 1>
33         [0..1] mouseMotionListenerField <argumentIsField: 1> <field>
34           [1..1] listenerField (String) <fieldName>
35           [1..1] typedMouseMotionListener <fieldOfType: 'MouseMotionListener'>
36           [1..1] initialized <assignedNew: 'void MouseMotionListener()' initializer: true>
37           [1..1] deregisters <callsTo: 'void Component.removeMouseMotionListener(MouseMotionListener)' location: 'void destroy()'>
38           ![1..1] field <argumentIsField: 1 sameAs: ../../listenerField>
39     [0..*] registersKeyListener <callsTo: 'void Component.addKeyListener(KeyListener)' position: 'after' location: 'void init()'>
40      !<1-1>
41         [0..1] this <argumentIsThis: 1>
42           [1..1] implementsKeyListener <assignableTo: 'KeyListener'>
43           [1..1] deregisters <callsTo: 'void Component.removeKeyListener(KeyListener)' location: 'void destroy()'>
44           ![1..1] this <argumentIsThis: 1>
45         [0..1] keyListenerField <argumentIsField: 1> <field>
46           [1..1] listenerField (String) <fieldName>
47           [1..1] typedKeyListener <fieldOfType: 'KeyListener'>
48           [1..1] initialized <assignedNew: 'void KeyListener()' initializer: true>
49           [1..1] deregisters <callsTo: 'void Component.removeKeyListener(KeyListener)' location: 'void destroy()'>
50           ![1..1] field <argumentIsField: 1 sameAs: ../../listenerField>
51     [0..*] Thread <field>
52       [1..1] thread (String) <fieldName>
53      ![1..1] typedThread <fieldOfType: 'Thread'>
54       [1..1] InitializesThread
55        !<1-1>
56           [0..1] initializesThreadWithRunnable <assignedNew: 'void Thread(Runnable)' position: 'after' location: 'void init()'>
57             <1-1>
58               [0..1] this <argumentIsThis: 1>
59                 [1..1] implementsRunnable <assignableTo: 'Runnable'>
60               [0..1] helper <argumentIsNew: 1 signature: 'void Runnable()'>
61               [0..1] variable (String) <argumentIsVariable: 1 signature: 'void Runnable()'>
62               [0..1] runnableField <argumentIsField: 1> <field>
63                 [1..1] typedRunnable <fieldOfType: 'Runnable'>
64                 [1..1] name (String) <fieldName>
65                 [1..1] initialized <assignedNew: 'void Runnable()' initializer: true>
66           [0..1] initializesWithThreadSubclass <assignedNew initializer: true subtypeOf: 'Thread'> <class>
67             [1..1] name (String) <fieldType> <fullyQualifiedName>
68             [1..1] overridesRun <methods: 'void run()'>
69             [1..1] extendsThread <assignableTo: 'Thread'>
70       [1..1] nullifiesThread <assignedNull location: 'void destroy()' position: 'after'>
71     [0..*] singleTaskThread <callsTo: 'void Thread(Runnable)' position: 'after' location: 'void init()' statement: true>
72       <1-1>
73         [0..1] runnable <argumentIsNew: 1 signature: 'void Runnable()'>
74         [0..1] runnableField <argumentIsField: 1> <field>
75           [1..1] typedRunnable <fieldOfType: 'Runnable'>
```

```
76          [1..1] name (String) <fieldName>
77          [1..1] initialized <assignedNew: 'void Runnable()' initializer: true>
78    [0..*] parameter <callsReceived: 'String Applet.getParameter(String)' location: 'void init()'>
79      [0..*] name <valueOfArg: 1>
80    [0..1] providesParameterInfo <methods: 'String[][] getParameterInfo()'>
81    [1..1] providesInfoForParameters <constraint: ../parameter implies: ../providesParameterInfo>
```

F.4 Metamodel of EJB FSML

```
1  EJBProject <project>
2    [0..1] InformationFromAnnotations
3      [0..*] BusinessInterface <class>
4        [1..1] interfaceName (String) <fullyQualifiedName>
5      [0..*] DerivedLocalInterface -|> BusinessInterface
6      [0..*] DerivedRemoteInterface -|> BusinessInterface
7      [0..*] ExplicitLocalInterface -|> BusinessInterface
8       ![1..1] localAnnotation <annotatedWith: 'Local'> <annotation>
9         ![1..1] isMarker <hasNoAttribute>
10     [0..*] ExplicitRemoteInterface -|> BusinessInterface
11      ![1..1] remoteAnnotation <annotatedWith: 'Remote'> <annotation>
12        ![1..1] isMarker <hasNoAttribute>
13     [0..*] EJBClass <class>
14       [1..1] className (String) <fullyQualifiedName>
15       [0..*] FieldAnnotatedWithEJB <field>
16         [1..1] fieldName (String) <fieldName>
17        ![1..1] EJBInterfaceAnnotation <annotatedWith: 'EJB'> <annotation>
18           [0..1] name (String) <attribute: 'name'>
19           [0..1] mappedName (String) <attribute: 'mappedName'>
20           [0..1] description (String) <attribute: 'description'>
21           [0..1] beanName (String) <attribute: 'beanName'>
22           [0..1] beanInterface (String) <attribute: 'beanInterface'>
23       [0..*] MethodAnnotatedWithEJB <method>
24         [1..1] methodName (String) <methodName>
25        ![1..1] EJBInterfaceAnnotation <annotatedWith: 'EJB'> <annotation>
26           [0..1] name (String) <attribute: 'name'>
27           [0..1] mappedName (String) <attribute: 'mappedName'>
28           [0..1] description (String) <attribute: 'description'>
29           [0..1] beanName (String) <attribute: 'beanName'>
30           [0..1] beanInterface (String) <attribute: 'beanInterface'>
31     [0..*] SessionBean -|> EJBClass
32       [0..1] localInterfaceSpecification <annotatedWith: 'Local'> <annotation>
```

```
33        ![1..*] localInterfaces <attribute: 'value'>
34           [1..1] interfaceName (String) <fullyQualifiedName>
35       [0..1] remoteInterfaceSpecification <annotatedWith: 'Remote'> <annotation>
36        ![1..*] remoteInterfaces <attribute: 'value'>
37           [1..1] interfaceName (String) <fullyQualifiedName>
38       [0..*] implementedLocalInterface <ImplementsExplicitLocalInterface>
39       [0..*] implementedRemoteInterface <ImplementsExplicitRemoteInterface>
40       [0..*] explicitLocalInterface <where attribute: interfaceName in: ../implementedLocalInterface>
41       [0..*] explicitRemoteInterface <where attribute: interfaceName in: ../implementedRemoteInterface>
42     [0..*] StatelessEJB -|> SessionBean
43      ![1..1] statelessAnnotation <annotatedWith: 'Stateless'> <annotation>
44         [0..1] name (String) <attribute: 'name'>
45         [0..1] mappedName (String) <attribute: 'mappedName'>
46         [0..1] description (String) <attribute: 'description'>
47     [0..*] StatefulEJB -|> SessionBean
48      ![1..1] statefulAnnotation <annotatedWith: 'Stateful'> <annotation>
49         [0..1] name (String) <attribute: 'name'>
50         [0..1] mappedName (String) <attribute: 'mappedName'>
51         [0..1] description (String) <attribute: 'description'>
52     [0..*] MessageDrivenEJB -|> EJBClass
53      ![1..1] messageDrivenAnnotation <annotatedWith: 'MessageDriven'> <annotation>
54         [0..1] name (String) <attribute: 'name'>
55         [0..1] mappedName (String) <attribute: 'mappedName'>
56         [0..1] description (String) <attribute: 'description'>
57     [0..*] Entity -|> EJBClass
58      ![1..1] entityAnnotation <annotatedWith: 'Entity'> <annotation>
59         [0..1] name (String) <attribute: 'name'>
60   [0..1] InformationFromDeploymentDescriptor <xmlDocument: '/META-INF/ejb-jar.xml'> <xmlElement name: 'ejb-jar'>
61     [0..*] DDSessionBean -|> DDBean <xmlElements: 'enterprise-beans/session'>
62     [0..*] DDNoTypeSessionBean -|> DDSessionBean <xmlElements: 'enterprise-beans/session'>
63      ![1..1] noSessionTypeElement <noXMLElement: 'session-type'>
64     [0..*] DDStatefulEJB -|> DDSessionBean <xmlElements: 'enterprise-beans/session'>
65      ![1..1] sessionType <xmlElements: 'session-type'> <xmlElement>
66        ![1..1] isStatefulSessionType <xmlElementValueEqualsString StringToSearchFor: 'Stateful'>
67     [0..*] DDStatelessEJB -|> DDSessionBean <xmlElements: 'enterprise-beans/session'>
68      ![1..1] sessionType <xmlElements: 'session-type'> <xmlElement>
69        ![1..1] isStatelessSessionType <xmlElementValueEqualsString StringToSearchFor: 'Stateless'>
70     [0..*] entityBeans <xmlElements: 'enterprise-beans/entity'>
71     [0..*] messageDrivenBeans <xmlElements: 'enterprise-beans/message-driven'>
72 DDBean <xmlElement>
73   [1..1] ejbName <xmlElements: 'ejb-name'> <xmlElement>
74     [1..1] ejbName (String) <xmlElementValue>
75   [0..1] ejbClass <xmlElements: 'ejb-class'> <xmlElement>
76     [1..1] ejbClass (String) <xmlElementValue>
```

```
77    [0..1] mappedName <xmlElements: 'mapped-name'> <xmlElement>
78      [1..1] mappedName (String) <xmlElementValue>
79    [0..1] description <xmlElements: 'description'> <xmlElement>
80      [1..1] description (String) <xmlElementValue>
```

References

[1] Struts applications project. http://sourceforge.net/projects/struts/. 56

[2] Enterprise JavaBeans deployment descriptor schema, May 2006. http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd. 37

[3] Struts 1.35 dtd, July 2006. http://struts.apache.org/1.3.5/dtds/struts-config_1_3.dtd. 37

[4] *Spring Framework Manual*, 2008. http://www.springframework.org/. 54

[5] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. *SIGPLAN Not.*, 40(10):345–364, 2005. 60

[6] Michal Antkiewicz. Round-trip engineering of framework-based software using framework-specific modeling languages. In *ASE*, pages 323–326, 2006. 5

[7] Michal Antkiewicz and Krzysztof Czarnecki. FeaturePlugin: feature modeling plug-in for eclipse. In *Eclipse Technology eXchange Workshop*, pages 67–72, 2004. 120

[8] Michal Antkiewicz and Krzysztof Czarnecki. Eclipse workbench part interaction fsml. Technical Report 2006-09, ECE, U. of Waterloo, 2006. http://gp.uwaterloo.ca/tr/2006-antkiewicz-wpi.pdf. 5

[9] Michal Antkiewicz and Krzysztof Czarnecki. Framework-specific modeling languages with round-trip engineering. In *MoDELS*, volume 4199 of *LNCS*, pages 692–706, 2006. 2, 5, 24

[10] Michal Antkiewicz and Krzysztof Czarnecki. Framework-specific modeling languages; examples and algorithms. Technical Report 2007-18, ECE, U. of Waterloo, 2007. v, 5, 27

[11] Michal Antkiewicz and Krzysztof Czarnecki. Design space of heterogeneous synchronization. In *GTTSE*, 2008. 25, 114

[12] Michal Antkiewicz, Krzysztof Czarnecki, and Matthew Stephan. Engineering of framework-specific modeling languages. *Transactions on Software Engineering*, 2008. Special Issue on Software Language Engineering. Submitted for review. Available from `http://swen.uwaterloo.ca/~mantkiew/2009-antkiewicz-engineering-fsmls.pdf`. v

[13] Michal Antkiewicz, Thiago Tonelli Bartolomei, and Krzysztof Czarnecki. Automatic extraction of framework-specific models from framework-based application code. In *ASE*, pages 214–223, 2007. 5, 15, 19, 49, 51, 52, 58, 59, 103

[14] Michal Antkiewicz, Thiago Tonelli Bartolomei, and Krzysztof Czarnecki. Fast extraction of high-quality framework-specific models from application code. *Journal of Automated Software Engineering*, 2008. Special Issue – Best Papers of ASE 2007. Accepted and recommended for publication. Available from `http://swen.uwaterloo.ca/~mantkiew/2008-antkiewicz-fast-extraction.pdf`. v

[15] Apache Software Foundation. *Roller Weblogger 3.0*. `http://rollerweblogger.org/`. 55

[16] Apache Software Foundation. *Struts User's Guide*. `http://struts.apache.org/1.3.8/index.html`. 52, 54, 55

[17] Apache Software Foundation. *A Walking Tour of the Struts MailReader Demonstration Application*. `http://svn.apache.org/viewvc/struts/struts1/trunk/apps/mailreader/src/main/webapp/tour.html?revision=481833`. 37

[18] Apache Software Foundation. *Struts 1.35 User Guide*, jul 2006. `http://struts.apache.org/1.3.5/userGuide/introduction.html`. 37

[19] Apache Software Foundation. *Struts Javadoc*, 2006. `http://struts.apache.org/1.3.5/apidocs/index.html`. 37

[20] Timo Asikainen, Tomi Mannisto, and Timo Soininen. A unified conceptual foundation for feature modelling. In *SPLC*, pages 31–40, 2006. 12

[21] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondȓej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Optimising AspectJ. In *PLDI*, pages 117–128, 2005. 112

[22] Don Batory. Feature models, grammars, and propositional formulas. Technical Report TR-05-14, University of Texas at Austin, 2005. 14

[23] J. Bosch, P. Molin, M. Mattsson, and P.O. Bengtsson. Framework - problems and experiences. In M. Fayad, D. Schmidt, and R. Johnson, editors, *Building Application Frameworks*. John Wiley, 1999. 6

[24] John Tang Boyland. Remote attribute grammars. *J. ACM*, 52(4):627–687, 2005. 117

[25] Martin Bravenboer and Eelco Visser. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In *OOPSLA*, pages 365–383, 2004. 115

[26] Aseem Paul Cheema. Struts2JSF - framework migration in J2EE using Framework-Specific Modeling Languages. Master's thesis, University of Waterloo, 2007. `http://hdl.handle.net/10012/3031`. 36, 105, 115

[27] E.J. Chikofsky and J.H. Cross II. Reverse engineering and design recovery: a taxonomy. *Software, IEEE*, 7(1):13–17, 1990. 111, 112, 114

[28] Code Generation Network. *CodeGeneration.net*. `http://www.codegeneration.net`. 113

[29] Tal Cohen, Joseph (Yossi) Gil, and Itay Maman. JTL: the Java tools language. In *OOPSLA*, pages 89–108, 2006. 111

[30] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming: methods, tools, and applications*. Addison-Wesley Publishing Co., 2000. 2, 7, 116

[31] Krzysztof Czarnecki, Simon Helsen, and Ulrich 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. 11, 14

[32] Krzysztof Czarnecki, Chang Hwan Peter Kim, and Karl Trygve Kalleberg. Feature models are views on ontologies. In *SPLC*, pages 41–51, 2006. 14

[33] Krzysztof Czarnecki and Peter Kim. Cardinality-based feature modeling and constraints: A progress report. In *Proceedings of the International Workshop on Software Factories, OOPSLA*, 2005. 11, 12, 120

[34] Krzysztof Czarnecki and Andrzej Wasowski. Feature diagrams and logics: there and back again. In *SPLC*, pages 23–34, 2007. 14, 120

[35] Kris De Volder. JQuery: A generic code browser with a declarative configuration language. In *PADL*, volume 3819 of *LNCS*, pages 88–102, 2006. 82, 111

[36] Jean-Marc DeBaud, Bijith Moopen, and Spencer Rugaber. Domain analysis and reverse engineering. In *ICSM*, pages 326–335, 1994. 111

[37] Linda DeMichiel. *EJB Core Contracts and Requirements*. Sun Microsystems, Inc., May 2006. 37

[38] Bill Dudney, Stephen Asbury, Joseph Krozak, and Kevin Wittkopf. *J2EE AntiPatterns*. Wiley, August 2003. 34

[39] Eclipse Foundation. *Eclipse documentation - Version 3.3: Editors*. http://help.eclipse.org/help33/index.jsp?topic=/org.eclipse.platform.doc.isv/guide/editors.htm. 37

[40] Eclipse Foundation. *Java Development Tools*. http://www.eclipse.org/jdt/. 28, 82, 113

[41] Eclipse Foundation. *Plug-in Development Environment*. http://www.eclipse.org/pde/. 29

[42] Eclipse Foundation. *Web Tools Platform*. http://www.eclipse.org/webtools/. 29

[43] Eclipse Foundation. *Java Emitter Templates Component*, 2007. http://www.eclipse.org/modeling/m2t/?project=jet. 113

[44] Eclipse Foundation. *Javadoc for Package org.eclipse.ui.part*, 2007. http://help.eclipse.org/help33/index.jsp?topic=/org.eclipse.platform.doc.isv/reference/api/org/eclipse/ui/part/package-summary.html. 37

[45] EJB 3.0 Expert Group. *JSR 220: Enterprise JavaBeansTM,Version 3.0*, 2006. http://java.sun.com/products/ejb. 54

[46] George Fairbanks, David Garlan, and William Scherlis. Design fragments make using frameworks easier. In *OOPSLA*, pages 75–88, 2006. 33, 55, 84, 110, 126

[47] Eduardo Figueiredo, Claudio Sant'Anna, Alessandro Garcia, Thiago Tonelli Bartolomei, Walter Cazzola, and Alessandro Marchetto. On the Maintainability of Aspect-Oriented Software: A Concern-Oriented Measurement Framework. In *CSMR*, 2008. 103

[48] Marcus Fontoura, Wolfgang Pree, and Bernhard Rumpe. Uml-f: A modeling language for object-oriented frameworks. In *ECOOP*, pages 63–82, 2000. 109

[49] Erich Gamma and Kent Beck. *Contributing to Eclipse: Principles, Patterns, and Plugins*. Addison-Wesley, 2003. 70, 84

[50] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. CodeQuest: scalable source code queries with datalog. In *ECOOP*, volume 4067 of *LNCS*, pages 2–27, 2006. 111

[51] Markku Hakala, Juha Hautamäki, Kai Koskimies, Jukka Paakki, Antti Viljamaa, and Jukka Viljamaa. Generating application development environments for java frameworks. volume 2186 of *LNCS*, pages 163–176, 2001. 109

[52] David Harel and Bernhard Rumpe. Meaningful modeling: What's the semantics of "semantics"? *Computer*, 37(10):64–72, 2004. 15

[53] Erik Hilsdale and Jim Hugunin. Advice weaving in AspectJ. In *AOSD*, pages 26–35, 2004. 59

[54] Marc R. Hoffmann. *Eclipse Workbench: Using the Selection Service*, April 2006. `http://www.eclipse.org/articles/Article-WorkbenchSelections/article.html`. 37

[55] D. Hou and H.J. Hoover. Using scl to specify and check design intent in source code. *Software Engineering, IEEE Transactions on*, 32(6):404–423, June 2006. 111

[56] Daqing Hou. *FCL: Automatically Detecting Structural Errors in Framework-Based Development*. PhD thesis, University of Alberta, 2004. 111

[57] Daqing Hou, Kenny Wong, and H. James Hoover. What can programmer questions tell us about frameworks? In *IWPC*, pages 87–96, 2005. 6

[58] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The unified software development process*. Addison-Wesley Longman Publishing Co., Inc., 1999. 3, 31

[59] Eric Jendrock, Jennifer Ball, Debbie Carson, Ian Evans, Scott Fordin, and Kim Haase. *The Java EE 5 Tutorial*. Sun Microsystems, Inc., September 2007. `http://java.sun.com/javaee/5/docs/tutorial/doc/index.html`. 37

[60] Ralph E. Johnson. Documenting frameworks using patterns. pages 63–76, 1992. 109

[61] Kyo Kang, Sholom Cohen, James Hess, William Nowak, and Spencer Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90TR -21, Software Engineering Institute, Carnegie Mellon University, 1990. 2, 7, 116

[62] U. Kastens, U. Kastens, W. M. Waite, and W. M. Waite. Modularity and reusability in attribute grammars. *Acta Informatica*, 31:601–627, 1994. 117

[63] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE Computer Society Press, 2008. 116

[64] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffry Palm, and William G. Griswold. An overview of AspectJ. In *ECOOP*, pages 327–355, 2001. 60

[65] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997. 59

[66] Chang Hwan Peter Kim. On the relationship between feature models and ontologies. Master's thesis, University of Waterloo, 2006. Available at `http://gsd.uwaterloo.ca/2006/05/11/peter-kims-masc-thesis/`. 14

[67] D. Kirk, M. Roper, and M. Wood. Identifying and addressing problems in framework reuse. In *IWPC*, pages 77–86, 2005. 6

[68] Barbara A. Kitchenham, Shari Lawrence Pfleeger, Lesley M. Pickard, Peter W. Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Trans. Softw. Eng.*, 28(8):721–734, 2002. 82

[69] Donald E. Knuth. Semantics of context-free languages. *Theory of Computing Systems*, 2(2):127–145, 1968. 116

[70] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, 1988. 109

[71] Hidehiko Masuhara and Kazunori Kawauchi. Dataflow pointcut in aspect-oriented programming. In *APLAS*, volume 2895 of *LNCS*, pages 105–121, 2003. 60

[72] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005. 116

[73] Ali Mesbah and Arie van Deursen. Migrating multi-page web applications to single-page ajax interfaces. *CSMR*, pages 181–190, 2007. 115

[74] William F. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, 1993. 115

[75] openArchitectureWare.org. *openArchitectureWare.* `http://www.openarchitectureware.org`. 113

[76] Alvaro Ortigosa and Marcelo Campo. Smartbooks: A step beyond active-cookbooks to aid in framework instantiation. In *TOOLS*, page 131, 1999. 109

[77] OSGi Alliance. *OSGi Service Platform, Core Specification, Release 4, Version 4.1*, 2007. `http://www.osgi.org/Specifications`. 28, 112

[78] Chinmay Pandit. *Make your Eclipse applications richer with view linking*, 2005. `http://www-128.ibm.com/developerworks/opensource/library/os-ecllink/`. 37, 52, 55

[79] Arnaud Payement. Type-based refactoring using JunGL. Master's thesis, Oxford University, 2006. 113

[80] Wolfgang Pree, Gustav Pomberger, Albert Schappert, and Peter Sommerlad. Active guidance of framework development. *Software - Concepts and Tools*, 3(16), 1995. 109

[81] Ruben Prieto-Diaz and G. Arango. *Domain Analysis and Software Systems Modeling*. IEEE Computer Society Press, 1991. 116

[82] Don Roberts and Ralph Johnson. Evolving frameworks: A pattern language for developing object-oriented frameworks. In *PLoP*, 1996. 115

[83] Atanas Rountev, Scott Kagan, and Thomas J. Marlowe. Interprocedural dataflow analysis in the presence of large libraries. In *CC*, volume 3923 of *LNCS*, pages 2–16, 2006. 112

[84] Spencer Rugaber. The use of domain knowledge in program understanding. *Annals of Software Engineering*, 9(1-4):143–192, 2000. 111

[85] A. Sellink, H. Sneed, and C. Verhoef. Restructuring of COBOL/CICS legacy systems. *CSMR*, pages 72–82, 1999. 115

[86] Dave Springgay. *Creating an Eclipse View*, November 2001. `http://www.eclipse.org/articles/viewArticle/ViewArticle2.html`. 37

[87] Sun Microsystems. *Java Server Faces*. `http://java.sun.com/javaee/javaserverfaces/`. 54

[88] Sun Microsystems. *Java Tutorials, Lesson: Applets*. `http://java.sun.com/docs/books/tutorial/deployment/applet/index.html`. 52, 54

[89] Sun Microsystems, Inc. *JavaTM Platform Enterprise Edition, v 5.0 API Specifications*, 2007. `http://java.sun.com/javaee/5/docs/api/`. 37

[90] Sun Microsystems, Inc. *Lesson: Applets*, February 2008. `http://java.sun.com/docs/books/tutorial/deployment/applet/index.html`. 10, 37

[91] Sun Microsystems, Inc. *Processes and Threads*, February 2008. `http://java.sun.com/docs/books/tutorial/essential/concurrency/procthread.html`. 37

[92] Tom Tourwé. *Automated Support for Framework-Based Software Evolution*. PhD thesis, Vrije Universiteit Brussel, 2002. 110

[93] Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing java bytecode using the soot framework: is it feasible? In *CC*, pages 18–34, 2000. 112

[94] Mathieu Verbaere, Ran Ettinger, and Oege de Moor. JunGL: a scripting language for refactoring. In *ICSE*, pages 172–181, 2006. 113

[95] Christopher Welty and Nicola Guarino. Supporting ontological analysis of taxonomic relationships. *Data Knowl. Eng.*, 39(1):51–74, 2001. 13

[96] Weilei Zhang and Barbara G. Ryder. Constructing accurate application call graphs for Java to model library callbacks. In *SCAM*, pages 63–74, 2006. 112