Engineering of Framework-Specific Modeling Languages

Michał Antkiewicz, Krzysztof Czarnecki, Matthew Stephan

Abstract—Framework-specific modeling languages (FSMLs) help developers build applications based on object-oriented frameworks. FSMLs model abstractions and rules of application programming interfaces (APIs) exposed by frameworks and can express models of how applications use APIs. Such models aid developers in understanding, creating, and evolving application code.
We present four exemplar FSMLs and a method for engineering new FSMLs. The method was created post-mortem by generalizing the experience of building the exemplars and by specializing existing approaches to domain analysis, software development, and quality evaluation of models and languages. The method is driven by the use cases that the FSML under development should support and the evaluation of the constructed FSML is guided by two existing quality frameworks. The method description provides concrete examples for the engineering steps, outcomes, and challenges. It also provides strategies for making engineering decisions.
Our work offers a concrete example of software language engineering and its benefits. FSMLs capture existing domain knowledge in language form and support application code understanding through reverse engineering, application code creation through forward engineering, and application code evolution through round-trip engineering.

Index Terms—framework-specific modeling language, domain-specific language, object-oriented framework, application programming interface, API, feature model, framework-specific model, forward engineering, reverse engineering, round-trip engineering, evolution, code pattern, mapping

1 Introduction

Object-oriented frameworks are widely used to develop applications in many domains. Frameworks provide *domain-specific concepts*, which are generic units of functionality. Developers create framework-based applications by writing *framework completion code* (also known as *application code*), which instantiates these concepts. For example, the framework underlying Eclipse's workbench offers concepts such as views and editors [1], [2]. Eclipse's outline view and Java editor are instances of these concepts. In order to create such instances, the developers 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. The API specifies the programming elements exposed to the user, e.g., the classes to subclass and 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 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 through an action contributor mechanism. Some implementation steps are different for a multi-page editor compared to a single-page one. In addition, 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 callback methods that are called by Eclipse's User Interface (UI) framework API must not be blocking.

API documentation, sample applications, and wizards, if available, offer some support in writing completion code. Cookbook-style articles and tutorials can be effective in demonstrating how framework-provided concepts should be instantiated; however, their main drawback is their passive nature, meaning that the developers must still 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 application code may be difficult to use as an implementation guide. In addition, 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 instantiation involves multiple API elements. Code generation wizards, as offered by some frameworks, are an active form of concept instantiation knowledge. Unfortunately, they usually cannot be re-

• *M. Antkiewicz, K. Czarnecki, and M. Stephan are with the Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, ON, N2L 3G1, Canada.*
E-mail: {mantkiew, kczarnec, mdstepha}@gsd.uwaterloo.ca

run with different settings once the generated code has been modified by a developer. 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 application developers can be significantly improved by taking a language perspective on framework APIs. In essence, a framework's API implicitly defines a domain-specific language, whose concepts are implemented and exposed through the mechanisms of the framework's programming language. The implementation is supplemented with documentation in natural language whenever the programming language is insufficient. For example, the concept of a Java Applet being a mouse listener [3] corresponds to several Java Applet API elements, including an API call to register a mouse listener and another call to deregister it. 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 application developers think about these higher-level concepts when programming, even though the concepts might not be explicitly represented by the programming language constructs.

Framework-specific modeling languages (FSMLs) [4], [5] are an explicit representation of the domain-specific concepts provided by framework APIs. FSMLs are used for expressing *framework-specific models* (FSMs) of application code. Such models describe the instances of framework-provided concepts that are implemented in the application code. In an FSM, each concept instance is characterized by a configuration of *features*. Features are concept properties such as single- or multi-page in the Eclipse editor example. Features correspond to *code patterns* that implement them in the application, such as classes implementing framework interfaces, calls to framework methods, and ordering of such calls.

We present our experience from building four FSMLs: *Eclipse Workbench Part Interactions (WPI) FSML*, *Java Applet FSML*, *Apache Struts FSML*, and *Enterprise Java Beans 3 (EJB) FSML*. The experience is presented as an *FSML engineering method*, which we devised post-mortem. Similar to the Unified Process of object-oriented software development [6], the method is *iterative* and *use-case-driven*. FSML use cases are central to the method since they embody the value proposition of FSMLs. Thus, FSML scoping and design decisions are made with respect to the use cases that the FSML under development should support. The method considers five main use cases in which FSMLs provide value to application developers: *framework API understanding*, *completion code understanding and analysis*, *completion code creation*, *completion code evolution*, and *completion code migration*. The method divides the life cycle of an FSML into iterations, each further subdivided into four phases: *inception*, *elaboration*, *construction*, and *transition*. Each phase involves several development *activities* and activities are subdivided into *steps*. The method adopts and extends activities from feature-oriented domain analysis [7]. The method description provides 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.

We ground the FSML approach in the *design science* paradigm [8] and a semiotic framework of model quality [9], [10]. Design science is a research paradigm in which "knowledge and understanding of a problem domain and its solution are achieved in the building and application of the designed artifact" [8]. The semiotic framework distinguishes between syntactic, semantic, and pragmatic qualities, and we apply it at the level of FSMs, FSMLs, and the entire *framework-specific modeling foundation* (FSMF). Some aspects of FSM and FSML qualities must be considered separately for each FSML; others depend on the FSMF and, thus, are inherent to the FSML approach. The assessment of the FSML-dependent qualities is part of the method. The method uses *Cognitive Dimensions* [11], [12] to asses the pragmatic quality of FSMLs.

The main contributions of this work are the four FSMLs and the experience of building them, packaged as an engineering method. The four languages constitute a set of *exemplars*, i.e., representative examples, of FSMLs. These exemplars demonstrate the ability of the FSML approach to support code understanding, creation, evolution, and migration for practical Java frameworks. The method offers FSML developers concrete steps and guidelines and is a necessary step in the maturation of the FSML approach. We make no claims about the quality of the method; however, the four FSMLs were evaluated as prescribed by the method.

The paper is organized as follows. We first describe how we applied the design science paradigm to develop the FSML approach (Section 2), followed by a brief overview of the key concepts and mechanisms of the approach (Section 3). We apply the model quality framework and Cognitive Dimensions to the different elements of the FSML approach in Section 4. We present the FSML engineering method in Section 5, followed by method justification in Section 6. We discuss the maturity of the different elements of the FSML approach and suggest directions for future work in Section 7. Finally, we discuss the related work in Section 8 and conclude the paper in Section 9.

2 Research Approach

The FSML approach was developed according to the *design science* paradigm [8], [13], which seeks to create innovations intended to solve a given problem using two processes: *build* and *evaluate*. These processes aid researchers in understanding the problem domain and in devising and checking feasibility of novel solutions. In contrast, the *behavioural science* paradigm, rooted in

natural science methods, takes a different approach: researchers develop theories, i.e., principles and laws, to explain and predict existing phenomena and then seek to justify these theories. Thus, the main processes in behavioural science are *theorize* and *justify*. Design science and behavioural science paradigms are complementary and inseparable. On the one hand, a successful artifact, as shown by evaluation, created using the design science paradigm may be subjected to theorize and justify to further the artifact's understanding. On the other hand, successful theories, as shown by justification, can be leveraged when building new artifacts in design science.

The design science paradigm categorizes artifacts as *instantiations*, *constructs*, *models*, and *methods*. For a given solution, instantiations are implementations of the approach, constructs constitute the conceptual foundation of the approach, models explain relationships among constructs, and methods explain the process of creating new instantiations. Often, instantiations are created prior to constructs, models, and methods.

To address the challenges of framework API usage, we built four FSMLs, each for a different framework and purpose. These FSMLs are the *instantiations* of the approach. The WPI FSML was built first and it was manually implemented. The Struts FSML was built next. It was also manually implemented, but it reused parts of the WPI FSML's implementation. Common parts of both FSML implementations were then factored out and generalized into an *FSML infrastructure*, which allowed FSMLs to be specified declaratively. The Applet FSML was the first language specified fully declaratively on top of the infrastructure. Eventually, all four languages were specified declaratively and the infrastructure was further generalized and refined during that process. Through that generalization, basic *constructs* and *models* of the FSML approach emerged. We refer to them as the framework-specific modeling foundation (FSMF). The FSML engineering *method* is the final artifact required by the design science paradigm; we present it in Section 5.

3 Framework-Specific Modeling Foundation (FSMF)

The main constructs of the FSMF are shown in Figure 1. The framework API (implicitly) provides a set of domain-specific concepts along with the constraints on their instantiations. The application code uses the API by implementing instances of these concepts. A concept instance is implemented through code patterns that adhere to the rules and constraints of the API. Code patterns can be structural (e.g., subclassing a framework class) or behavioural (e.g., calling a framework method in the control flow of an object, order of method calls) [14].

An FSML explicitly models the concepts and constraints prescribed by the framework API as *feature models*. Feature models represent concepts as hierarchies of *features*, which are distinguishing characteristics among

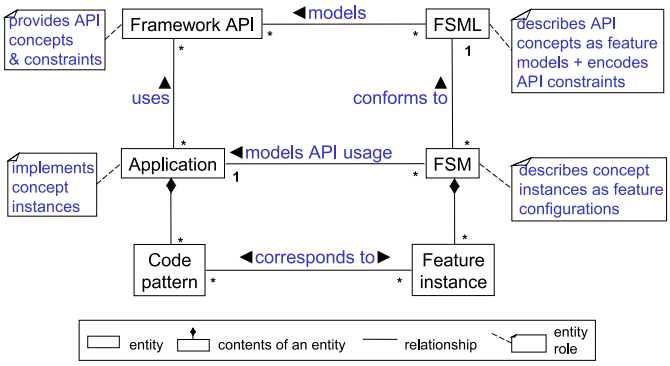


Fig. 1. Modeling framework API usage by FSMs

concept instances [7]. In particular, we use *cardinality-based feature models* [15], [16], which support multiple instantiations of features, feature attributes (including feature reference attributes), additional constraints over features (such as implies or excludes), and feature inheritance. The distinction between a concept and a feature is a matter of focus: the root of the feature hierarchy within the current scope of consideration denotes a concept, whereas its children denote features of that concept. When the scope changes, a feature may become a concept and vice versa [17, apdx. A]. Thus, any node in a feature hierarchy can be seen as a concept for all of its descendents, and any node except the root can be seen as a feature of any of its direct or indirect parents.

Effectively, feature modeling is the abstract syntax definition formalism for FSMLs. Other choices such as class models or textual grammars are also possible; however, feature models are particularly well suited because of their ability to concisely represent commonalities and variabilities and their wide use in domain analysis [7], [17, ch. 4]. The suitability of feature modeling to domain analysis is important since a substantial part of FSML engineering can be understood as domain analysis.

An FSM models the API usage of the application. It describes concept instances as *feature configurations*, which consist of *feature instances* configured according to the constraints imposed by the FSML's defining feature model. Each feature instance corresponds to one or more code patterns that implement the feature instance in the application code. An FSM can help application developers answer questions such as how the application is using the framework, what concepts the application implements, whether the application uses the framework correctly, and how the application should use the framework.

Importantly, features may represent both high-level domain concept properties, such as single- or multi-page for the Eclipse editor example, and low-level implementation variants, such as adding an action to the editor's toolbar directly or through a contributor. The high-level features are usually defined by aggregating or referencing lower-level features in the feature model or both. Consequently, the code pattern implementing a

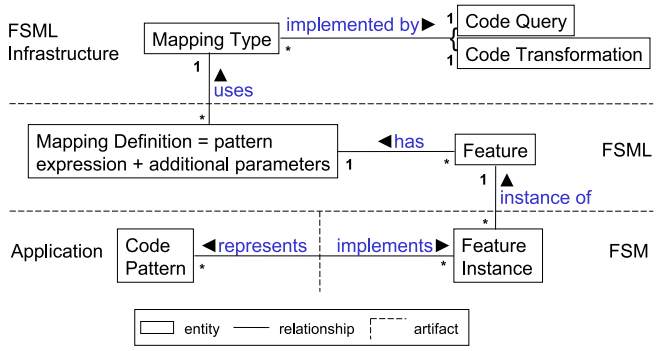


Fig. 2. Defining feature-instance-to-code-pattern correspondence using mapping definitions

high-level feature is a composition of the code patterns corresponding to the lower-level features that it aggregates or references.

Figure 2 illustrates how the correspondence between feature instances and code patterns is established. The figure is divided into four parts, each representing a different artifact: application, FSM, FSML, and FSML infrastructure. A feature instance in an FSM represents one or more code patterns in the application. Instances of the relationship between the code patterns and feature instances are referred to as *traceability links*. The feature-instance-to-code-pattern correspondence is specified as part of the FSML definition by attaching *mapping definitions* to features. Each mapping definition contains a *pattern expression* that specifies the desired code patterns using a *mapping type*. Mapping types are generic and reusable specifications of feature-instance-to-code-pattern correspondences that are provided by the FSML infrastructure. For example, there are mapping types for specifying the correspondence to Java classes that are compatible with a given type, calls to methods with a given signature, or XML elements on a given path. Mapping types define *parameters*, such as the method signature or the XML element path, which are set in pattern expressions.

Since the mapping types are only specifications, they are implemented by *code queries* for detecting the specified code patterns in the application code and *code transformations* for adding the code patterns to or removing them from the code. Effectively, code queries enable *reverse engineering* (RE), i.e., the automatic retrieval of FSMs from application code by detecting feature instances in the code [14]. Code queries are executed for features in the FSML's metamodel. If a code pattern is matched, an instance of the given feature is added to the FSM and a traceability link is established. Traceability links allow navigating from feature instances in the model to code patterns and vice versa. Analogously, code transformations enable *forward engineering* (FE), i.e., the generation of code from FSMs by successively executing, for each of their feature instances, the transformations for code pattern addition. Finally, both code queries and code transformations enable *round-trip engineering* (RTE), i.e., the ability to propagate modifications of the application code to its model and vice versa [18]. In RTE, a *prescriptive FSM*, i.e., a model of how an application should use an API, is first compared with the *descriptive FSM*, i.e., the FSM of how the current application code actually uses the API. The descriptive model is obtained through RE, and the comparison uses the last-reconciled model as a reference, i.e., we use a 3-way compare [19]. The last-reconciled model records the common base of the prescriptive and the descriptive models from the time of their last reconciliation. The reference allows us to determine whether a feature instance present only in one of the two models, i.e., the descriptive or the prescriptive one, was added to one them or removed from the other. After the comparison, the user can review the differences and resolve conflicts, if any. As a last step, the prescriptive model and the code are updated. The prescriptive model is updated by copying the new features from the descriptive model and/or removing the features that are not present in the descriptive model. The code is updated by executing code transformations that add, remove, or modify feature implementations in the code. Currently, code transformations support only code addition. If the code transformations are not implemented, RTE can still be used for comparing the code and the model and incrementally updating the model.

In general, the queries and transformations of the mapping types for specifying behavioural patterns can only be approximations of the respective mapping type semantics. This limitation is due to the undecidability of dynamic properties of programs. For code queries, approximation means the possibility of *false negatives*, i.e., feature instances undetected by the query but present in the code; and *false positives*, i.e., feature instances detected by the query but absent from the code. The quality of approximation can be measured using *precision* and *recall* [20]. Precision is related to false positives: the fewer false positives, the higher the precision. Recall is related to false negatives: the fewer false negatives, the higher the recall. A perfect approximation has both precision and recall of 100%. Our previous work showed the feasibility of devising code queries that were able to detect all the behavioural features of three exemplar FSMLs in a large body of open-source application code with very few false negatives (high recall) and false positives (high precision) [14]. The exemplars considered in that study were the WPI, Applet, and Struts FSMLs. Code queries and transformations may define *additional parameters*, i.e., parameters that are needed in addition to those defined by the corresponding mapping types. These parameters may be set as part of mapping definitions in order to tune the behavior of the queries and transformations. For example, query parameters may determine the context-sensitivity of the code analysis, and transformation parameters may specify the preferred location for a method call to be added.

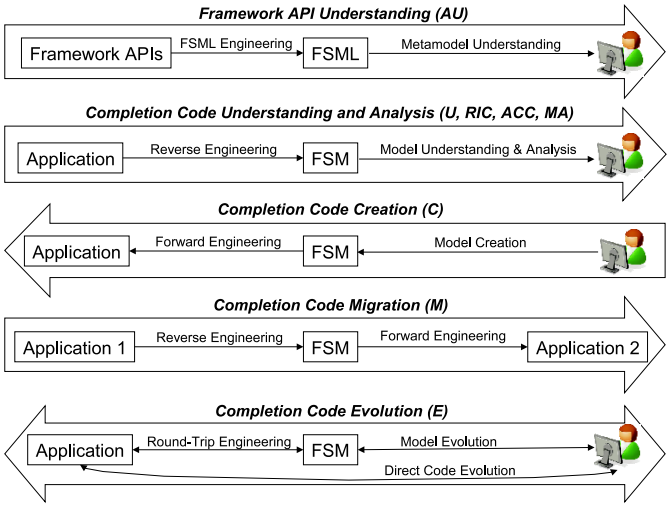


Fig. 3. Overview of FSML Use Cases

The generic FSML infrastructure delegates the execution of code queries and transformations to *pluggable mapping interpreters*. This mechanism simplifies the implementation of new mapping types that may be required by a particular FSML and it allows support for multiple artifact types, such as Java, XML, and C/C++. The related Ph.D. thesis [5] contains details on the technical aspects of the infrastructure: its architecture and implementation.

The FSML approach was designed to support five use cases illustrated in Figure 3.

1) **Framework API understanding (AU)** through manual inspection of the concepts and mapping definitions of the FSML metamodel.
2) **Completion code analysis and understanding** through RE. This use case groups four more specific use cases:
   a) *understanding* (U) by inspecting the FSM and navigating to code patterns through traceability links;
   b) *referential integrity checking* (RIC) by evaluating model queries over the FSM to check whether consistent values are used throughout the code, possibly throughout multiple artifacts of different types such as Java and XML;
   c) *API constraint checking* (ACC) by evaluating cardinality and additional constraints over the FSM; and
   d) *model analysis* (MA) by analyzing or transforming the FSM 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.
3) **Completion code creation (C)** by generating code for an existing FSM through FE. Creation involves incrementally adding new code patterns to the existing code; however, it is the developer's responsibility to manually resolve possible conflicts, such as duplicate methods or classes. Manual conflict resolution is necessary because FE does not take into account existing code as RTE does.
4) **Completion code migration (M)** between framework versions or even different but conceptually similar frameworks through RE and FE. 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 through applying RE to the code that uses the source framework and executing specialized FE to produce the code that uses the target framework. Both frameworks need to be conceptually close since a single model is used. Otherwise separate source and target FSMLs with a model transformation are required.
5) **Completion code evolution (E)** by independently evolving and synchronizing the completion code and the FSM through RTE. Evolution of the code can be observed by comparing FSMs extracted at different points in time. The code can be evolved by first changing its corresponding FSM and then propagating the changes to code using RTE. Unlike FE, RTE first identifies feature instances that are already implemented and only adds code patterns related to new feature instances.

We will provide concrete examples of feature models, mapping types, and mapping definitions in Section 5.

4 Artifact Qualities in the FSML Approach

Following the design science paradigm, any novel artifact built (constructs, models, methods, and instantiations) must be evaluated with respect to its goals. In particular, the quality of the FSMLs being engineered needs to be measured and such measurement needs to be part of an FSML engineering method. This section presents a framework for defining the quality of models and languages, which we use to define the quality of FSMs, FSMLs, and the FSMF. The framework builds on prior work on *model quality* [9], [10] and *Cognitive Dimensions* for evaluating notations [11], [12].

4.1 Model Quality

Lindland et al. proposed a general semiotic framework for defining the quality of models [9]. Their basic assumption, rooted in semiotics (e.g., [21]), is that a model is expressed in some language, represents some domain, and has some audience (cf. Figure 4). In this setting, three basic types of model quality can be considered [9].

Syntactic quality is the extent to which the model is well-formed with respect to the modeling language.

Semantic quality is the extent to which the model is *valid* and *complete* with respect to the domain. Validity means that all statements made by the model are correct about and relevant to the domain. Completeness means

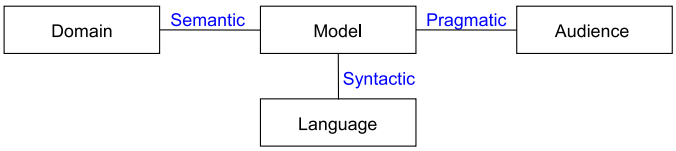


Fig. 4. General Quality Framework [9]

that the model makes all the statements about the domain that are correct and relevant. The relevancy of the statements depends on the scope of the model: the real-world domain being modeled may contain properties that are irrelevant and, thus, outside the scope of the model.

Pragmatic quality is the extent to which the model can be constructed, comprehended, and modified by its audience. More precisely, the comprehension goal of pragmatic quality is the complete and correct understanding of the model by its interpreting audience. Furthermore, a model created or modified by an audience should completely and correctly reflect the intention of the audience. Pragmatic quality depends on many model properties that impact the ease of creation, comprehension, and modification, such as visual appearance (e.g., use of diagrams or text, meaningful icons and names, and quality of layout), conciseness, and closeness of mapping (i.e., the extent to which the structure of the domain is explicit in the model). These properties are highly interdependent. For example, conciseness can be improved at the cost of explicit structure by introducing shorthands.

Lindland et al. introduced the notion of *feasibility* of quality goals [9]. Intuitively, feasibility means that the benefits of improving the quality with respect to a given goal should be greater than the drawbacks of doing so. For semantic quality, *feasible validity* is reached when the benefits from removing an invalid statement from the model are less than the drawbacks; *feasible completeness* is reached when the benefits of adding new statements is less than the drawbacks. Drawbacks may involve both economic issues and factors such as user preference and ethics. For pragmatic quality, *feasible comprehension* is reached when the benefits from correctly comprehending statements in the model that have not yet been considered or were misunderstood are less than the drawbacks. Similarly, *feasible construction* and *feasible modification* balance benefits and drawbacks by relaxing the requirement that a newly constructed or modified model completely and correctly reflects the intention of the audience.

Lindland et al. also distinguish between qualities and the *means* for improving the qualities. For example, syntactic quality can be improved by means of error prevention (such as structured editing), error detection (syntax checking), and error correction (such as automatic correction proposals). In our setting, means for improving semantic quality include model consistency checking, RE, FE, and RTE. Finally, means for improving

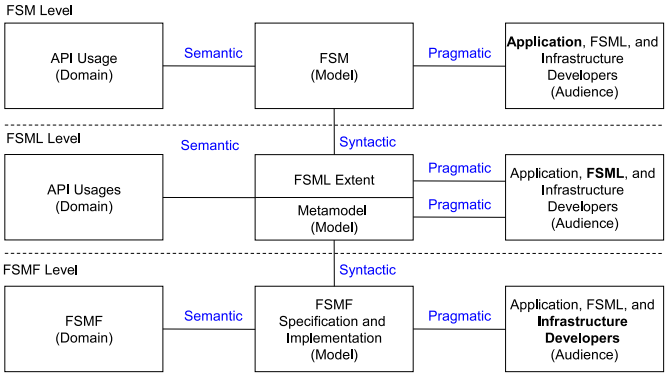


Fig. 5. Quality framework applied at each of the three levels of the FSML approach

pragmatic quality include visualization, filtering, explanation, animation and simulation, and refactoring.

In addition to these three basic qualities, we also consider the *organizational quality* of a model, which is the extent to which the model fulfills the goals of the organization that is using it [10]. Organizational quality reflects the ultimate value of a model to an organization. This quality depends on the syntactic, semantic, and pragmatic qualities of the model, but it also considers social aspects of an organization, such as the diversity of its members who use the model and the rules and practices of the organization.

4.2 Quality of FSMs, FSMLs, and the FSM Foundation

We apply the model quality framework at three levels: FSMs, FSMLs, and the FSM foundation (cf. Figure 5). The domain of an FSM is a particular API usage that the model represents. The primary audience for FSMs are application developers, but FSML developers and FSML infrastructure developers also need to read and create FSMs as part of their work (the primary audience of each artifact is typed in bold face in Figure 5). As expected, the syntactic quality of an FSM is defined with respect to its FSML; the semantic quality is defined with respect to the API usage it represents; and the pragmatic quality is defined with respect to its audience. These three quality types are defined along the same lines for an FSML, except that the domain is a set of API usages; the syntax is defined by the specification of the FSM foundation; and the primary audience are FSML developers.

We distinguish between the pragmatic quality of the FSML extent, i.e., the set of valid FSMs, and the pragmatic quality of the metamodel, i.e., the FSML definition. The reason is that, in general, the same FSML extent can be defined by many different metamodels, e.g., each using inheritance differently, and each such metamodel has its own pragmatic quality. The pragmatic quality of the extent can be thought of as the cumulative pragmatic quality of the models in that extent. Given the semantics to be expressed, the language enables achieving a

certain level of pragmatic model quality. In general, if a language contains *representational redundancy*, i.e., if there is more than one way to express the same semantics, a particular model may not achieve the best pragmatic quality, e.g., conciseness, that would be possible for the given semantics. In FSMLs, since the mapping from API usages to FSMs is a function, there is no representational redundancy: given a particular API usage, the pragmatic quality of the extent directly translates into the pragmatic quality of a given model. Obviously, the pragmatics of the metamodel and the pragmatics of the extent overlap.

The pragmatic quality of FSMLs can be assessed using the Cognitive Dimensions (CDs) framework [11], [12]. The framework takes a whole-system view: it assesses the quality of a *system* consisting of the notation and its *environment*. The environment is essentially the means used to manipulate representations in the notation, i.e., the modeling means in our context. The framework defines the following dimensions.

Abstraction Gradient reflects whether a notation disallows, allows, or even requires defining new abstractions when creating a new model. The notation is then referred to as *abstraction-hating*, *abstraction-tolerant*, or *abstraction-hungry*, respectively. Abstraction mechanisms add to the weight of a language in terms of learning and usage cost. In particular, abstraction hungry languages suffer from "delayed gratification" because the appropriate abstractions must be defined before the programmer's inherent goals can be addressed [12]. FSMLs are abstraction-hating: an FSM instantiates only the language-provided abstractions (concepts and features) and no new abstractions can be created as part of the FSM.

Closeness of mapping represents the similarity of the structures in the domain to the structures in the model. In general, FSMLs can represent both hierarchies of API-related application code elements and structures that cross-cut such hierarchies; however, closeness of mapping needs to be assessed for individual FSMLs as part of the FSML engineering method.

Consistency is related to the "guessability" of a notation. When a subset of the language has been learnt, how much of the remainder can be inferred? For FSMLs, consistency translates into the use of consistent and meaningful criteria for naming, nesting, and grouping of features in the FSML metamodel. This dimension needs to be assessed as part of the method.

Terseness is related to the number of model elements needed to express a meaning. FSMLs promote terseness since FSMs contain only concept and feature instances without any superfluous syntax. Support for computing default feature selections can help improve this dimension for FE and RTE by allowing the modeler to state only the features of interest explicitly.

Error-proneness is related to the question whether the notation induces mistakes. It needs to be assessed for individual FSMLs as part of the method.

Hard mental operations refers to the existence of mental operations at the notation level that are difficult to perform, such as certain combinations of constructs being difficult to understand. For example, if a property of interest is represented by several features scattered across an FSM rather than a single feature, inferring the feature may require resorting to paper and pencil. This dimension needs to be assessed for individual FSMLs as part of the method.

Hidden dependencies are dependencies among model elements that are implicit in the model and are difficult to uncover. The hierarchical structure of concepts is explicit and most visible. Dependencies created through feature attributes referring to other features are less visible and require some means to support their traversal. The most hidden dependencies are incurred by the additional constraints defined over two or more features, such as implications and mutual exclusions. This dimension is assessed for each FSML as part of the method. Interestingly, the vast majority of dependencies in our four exemplar FSMLs were encoded in feature hierarchies, and only relatively few reference attributes and additional constraints were needed.

Premature commitment refers to situations where modelers have to make decisions before they have the information they need, e.g., when information has to be specified in a strict order. The natural construction order for concept instances in an FSM is top-down; however, a feature configuration interface can additionally support bottom up configuration by choice propagation. Consequently, FSMLs do not require premature commitment.

Progressive evaluation refers to the ability to build a model incrementally. FSMLs support the incremental creation of models. In particular, incremental code addition and round-trip engineering allow testing the result of adding new concept and feature instances.

Role-expressiveness refers to the ability to recognize the role of an element in a model and is related to self-describability of the notation. The role-expressiveness of a vanilla FSM is relatively low since all elements are concept or feature instances; however, this quality can be improved by annotating concepts and features with feature types indicating their roles, e.g., components, connectors, API constraints, high-level domain properties, and implementation variants. Our exemplar FSML implementations use custom icons to make these roles visible in FSMs. Assessing this dimension is part of the method.

Secondary notation refers to the ability to use layout, colour, or other cues to convey extra meaning above and beyond the official semantics of the language. Support for comments and annotations can be easily provided.

Viscosity is related to the amount of effort required to perform a given change. The presence of global dependencies such as references and cross-cutting constraints increases viscosity. Viscosity can be reduced by improving modularity, i.e., increasing cohesion and reducing coupling. The exemplar FSMLs have low viscosity with respect to feature re-configuration as they have only few references and constraints. FE and RTE help reduce the

viscosity of FSMs in the presence of application code.

Visibility is related to the ability to see all the relevant information simultaneously (assuming a large enough display). Visibility can be improved by providing filtering facilities.

Some of the dimensions, such as abstraction gradient, are specific to the entire approach; other need to be assessed for individual FSMLs as part of the FSML engineering method. The dimensions to be assessed on an FSML-basis are closeness of mapping, consistency, error-proneness, hard mental operations, hidden dependencies, role-expressiveness, viscosity, and visibility.

Finally, the quality of the entire approach depends on the quality of the FSM foundation (cf. Figure 5). For the sake of our discussion, we take the Platonic stance that an ideal FSMF exists and only waits to be discovered and explicitly represented. This stance allows us to consider the qualities of the FSMF specification and implementation, such as the degree of formality (not shown in the figure), semantic correctness, and usability.

4.3 Quality of RE, FE, and RTE

The purpose of modeling means is to improve model quality. Thus, we also need to consider the quality of the key modeling means in our context, namely RE, FE, and RTE. These means rely on a set of algorithms which are part of the foundation [5]. These algorithms have been tested using the exemplar FSMLs.

The RE algorithms are designed such that they can only produce well-structured models modulo possible violations of cardinality and additional constraints. Such violations are allowed since RE should also be able to produce models of applications that use an API incorrectly. The semantic quality of RE depends on the correctness of the mapping definitions and code queries implementing the corresponding mapping types. Since the queries are approximations of the mapping definitions, our goal is to achieve *feasible* semantic quality. The mapping definitions and queries are tested for the presence of false positives and false negatives on sample applications as part of the method and improved as needed. The improvement may require modeling features differently, adjusting their mapping definitions, or even implementing new queries. RE has no impact on the model pragmatics since there is only one correct model for a given application.

FE relies on the mapping definitions and code transformations implementing the corresponding mapping types in order to create syntactically and semantically correct code. Thus, the mapping definitions and the transformations need to be tested for different input models as part of the method. Depending on the application context, the generated code may or may not be complete; thus, our goal is *feasible* completeness. In addition to syntactic and semantic correctness needed to achieve *feasible* validity, other qualities of the generated code, such as performance and readability, also need to be assessed as part of the method. As a result of the assessment, the FSML may need to be refined, e.g., by adding new feature implementation variants.

RTE relies on the quality of the mapping definitions and their queries and transformations in a similar way as RE and FE do; however, RTE places additional requirements on the transformations, as they should be able not only to add implementations of concept and feature instances to an existing application, but also remove or modify implementations. Currently, the FSML infrastructure provides only transformations for adding implementations. That is, removals and modifications need to be done manually at the code level; however, the infrastructure eases this process by providing full traceability between the model and the code. Furthermore, RTE relies on the quality of matching between the currently asserted prescriptive model and the descriptive model retrieved from the application code. The matching relies on the definition of concept and feature keys, which is part of the method.

4.4 Organizational Quality of FSMLs

The organizational quality of an FSML is the extent to which the FSML fulfills its stated use case goals (cf. Figure 3). This quality relies on qualities of the involved artifacts and means.

For framework API understanding, application developers rely on the syntactic, semantic, and pragmatic quality of the metamodel.

For completion code understanding and analysis, application developers rely on the semantic completeness of the FSML (i.e., the ability to express all relevant information), the pragmatic quality of the FSML extent (for model comprehension), and the quality of RE (only few false negatives and/or false positives).

For completion code creation, application developers rely on the semantic and pragmatic quality of FSMLs, the quality of code generation (FE), and the syntactic and semantic quality of FSMs. Means such as editing assistance (e.g., auto-completion) and consistency checking can improve syntactic and semantic quality of FSMs.

For completion code migration, application developers rely on the same qualities as those for code understanding and analysis and for code creation.

For completion code evolution, application developers additionally rely on the quality of model comparison and code transformation.

FSML developers also rely on the pragmatic quality of the metamodel in order to evolve it.

Finally, application, FSML, and infrastructure developers ultimately depend on the quality of the FSMF.

5 FSML Engineering Method

This section presents the FSML engineering method. The method was constructed post-mortem by generalizing

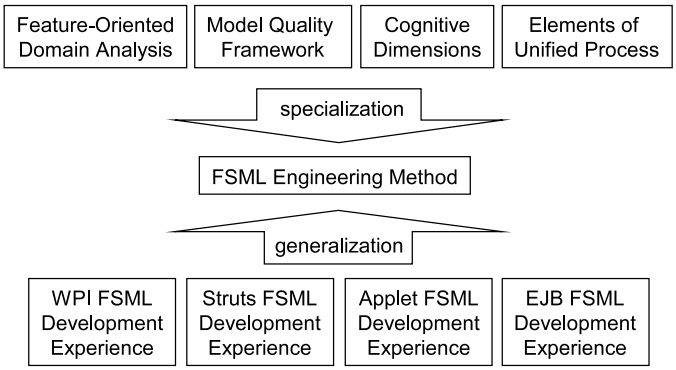


Fig. 6. Inputs to method construction

the experience of building four FSMLs and by specializing a set of well-established, general approaches (cf. Figure 6): *Feature-Oriented Domain Analysis* (FODA) [7], [17, ch. 4], the model quality framework [9], the Cognitive Dimensions framework [11], [12], and a few elements of the *Unified Process* [6]. We refer to these general approaches as the *base approaches* of the method.

Constructing the method allowed us to elicit and organize our experience in a systematic way. The base approaches provided the overall method structure and the generic concepts and steps that were used to drive the elicitation. FODA was selected as a basis for the domain scoping and feature modeling activities. The model quality and the Cognitive Dimensions frameworks were used to establish the evaluation criteria for FSMLs as described in Section 4. The overall iterative life cycle model and the idea of use-case driven development were borrowed from the Unified Process. This choice was motivated by the fact that the exemplar FSMLs were developed in an iterative and use-case driven fashion.

The exemplar FSMLs were developed outside of a controlled environment since their development drove the creation of the FSML foundation and infrastructure as described in Section 2. Thus, we relied on retrospection and inspection of the current and past versions of the FSMLs to provide concrete examples of engineering steps and issues. The elicitation of these steps and issues involved specializing the base approaches to the FSML context, e.g., extending FODA to handle cardinality-based feature modeling and mapping definitions, while identifying corresponding examples in the FSML metamodels and recalling the issues that arose during their creation.

Since the method embodies our experience with FSML engineering, we make no claims about the method's completeness or generality. Instead, we justify the engineering steps with examples from the exemplar FSMLs.[1] Furthermore, Section 6 provides more information on how well the base approaches fit the actual experience and how they had to be specialized and adapted. Section 6 also gives ideas of how the method may be specialized during its application.

1. The complete metamodels of the FSMLs can be found elsewhere [5], [22].

5.1 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.

The overall structure of our engineering method was inspired by the Unified Process for software development [6]. The life cycle of an FSML consists of multiple iterations, each having the following phases as shown on Figure 7. Each iteration has different focus and requirements and the goal of each iteration is to deliver a working increment in functionality to the users.

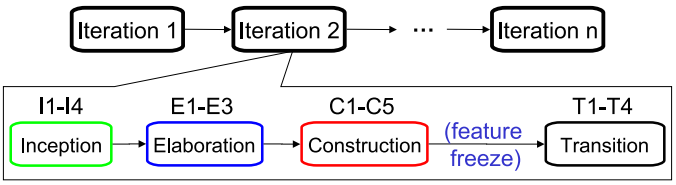


Fig. 7. FSML life cycle: iterations and phases

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 for the current iteration.
2) *Elaboration* is focused on identifying the main concepts and creating the overall structure of feature models.
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 followed directly by FSML developers. These instructions are marked by [!]. We also mark questions that should be answered by a step or activity by [?]. The presentation uses a naming scheme whereby each activity is labeled

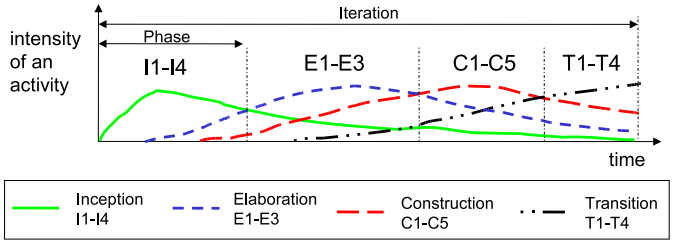


Fig. 8. FSML life cycle: phases and their dominant activities in a single iteration

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 8 illustrates the intensity of performing different activities in different phases of the FSML's life cycle. The figure suggests that the activities and steps need not be performed in the order of their presentation in the method. Furthermore, not every step and activity are performed in every iteration.

5.2 Inception

Inception takes as input a concrete framework for which an FSML is to be designed or a framework and an existing FSML to be refined. Table 1 shows information related to the inception of the four exemplar languages. The first row shows the inception date in the first iteration for each language. In this phase, the following activities are performed.

I1: Determine the purpose of the language.
? What problem or problems should the language address and what is the value proposition for this iteration?
! Value proposition should justify the effort of performing the iteration.

Table 1 summarizes the value propositions for the four exemplar FSMLs. The main motivation for developing the WPI FSML was that implementing Eclipse workbench part interactions involves several implementation steps that are usually highly scattered in the completion code. Thus, determining the presence and the properties of interactions in the completion code was challenging for application developers. Also, some necessary implementation steps could be easily missed when implementing interactions. Consequently, the value proposition of the WPI FSML was 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 by offering round-trip engineering.

The Struts FSML was initially developed to demonstrate the feasibility of automatic migration of Struts applications to the 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, the focus of the Struts FSML shifted toward 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 concept instances are represented both in Java and XML and they are related by name. This naming convention increases the possiblity of naming mistakes that are typically discovered only at runtime and thus reducing productivity. The FSML allows detecting such mistakes at development time.

The Applet FSML was developed primarily as a 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. This language was the main example used when extending our FSML infrastructure with support for declarative mapping definitions. It was also used to compare the FSML approach with the design fragments approach, which documents usage patterns for framework APIs [23]. 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 were challenges caused by the introduction of Java annotations as an additional configuration mechanism in EJB 3.0. The new mechanism 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 to understand how it originated. The EJB FSML offers exactly such a tool. Furthermore, a secondary purpose of the FSML is to detect EJB antipatterns, such as *Bloated Session* or *Fragile Links* [24].

I2: Determine which use cases should be supported.
? Which use cases are required to achieve the language's purpose in the current iteration?
! Consider the use cases illustrated in Figure 3 and described in Section 3.

The use cases supported by each exemplar FSML are specified in Table 1.

Framework API understanding (AU). 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, and they also can be used to learn about these aspects.

Completion code understanding (U). All four exemplars support this use case.

Referential integrity checking (RIC). Struts FSML contains features that correspond to information from Java and XML and features that correspond to the referential integrity constraints.

TABLE 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 on (1) semi-automatic completion code migration from Struts to Java Server Faces; later the focus changed to (2) referential integrity checking between Java and XML and (3) visualizing web-page flow. Last (4) code creation was added.	(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	U, ACC, C, E	M, later U, RIC, last C	AU, U, ACC, C, E	U, RIC, ACC, MA
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

API constraint checking (ACC). All exemplars except Struts FSML support this use case (Struts FSML focuses solely on checking referential integrity).

Model analysis (MA). In EJB FSML, the analysis involves applying override rules to produce a run-time view of the EJB configuration.

Completion code creation (C). WPI, Struts, and Applet FSMLs support incremental code creation. Support for this use case was later added to Struts FSML since all code transformations for the used mapping types were already implemented.

Completion Code Migration (M). Both Struts and JSF frameworks have similar concepts but different ways of implementing them and therefore the completion code can be automatically migrated [25]. The first version of the Struts FSML supported this use case.

Completion Code Evolution (E). WPI and Applet FSMLs support RTE for all of their features.

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

? Which sources provide information about the concepts and features in the scope?

! Consider the following items as possible sources of framework knowledge: API documentation, tutorials, articles, expert knowledge, experience, sample applications, and existing metamodels and XML schemas provided with the framework.

Table 2 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 total number of features in the language that originated from the given source. A summary of features originating from all documents is given in the column ΣDoc. Features originating from non-documentation sources are presented in columns to the right of the column ΣDoc. The last column gives the total number of features for each language.

A significant percentage of the features in WPI and Applet FSMLs were created from experience (Table 2). These statistics reflect the fact that the developer of these FSMLs had previous experience in using the corresponding frameworks. In contrast, the developers of the Struts and EJB FSMLs did not have prior experience with the frameworks; thus, the vast majority of the features were extracted from documentation. The distribution of features over their documentation source types largely depends on the availability and quality of the corresponding sources. For frameworks that have extensive configuration schemas, such as Struts and EJB, the schemas were a significant source of knowledge, e.g., 28% of features for Struts and 18% of features for EJB. Such schemas can be viewed as metamodels of the artifacts they represent and can be, to some degree, incorporated into the FSML. For the sample FSMLs, only two features (for Applet FSML) were obtained from example applications (EA); however, their subfeatures were later added based on API documentation. Also, the mapping definitions of many features were improved based on the analysis of example applications [14].

I4: Determine the scope of the language.

? What concepts and features are in the scope of the language?

! Consider horizontal and vertical scope.

Horizontal scope is measured with respect to the coverage of the API. 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

TABLE 2
Distribution of features over sources of knowledge

	Source kind	API		Tutorials		Articles			ΣDoc	Other (non-doc)				ΣF
WPI	**Source**	**DW [26]**		**TW_1 [1]**		**AW_1 [2]**	**AW_2 [27]**	**AW_3 [28]**	**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 [29]**	**SS [30]**	**TS_1 [31]**		**AS_1 [32]**			**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**			**TA_1 [33]**	**TA_2 [34]**				**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 [35], [36]**	**SE [37]**	**TE_1 [38]**					**D***	**I**				**F***
	Features	21	13	27	-	-	-	-	61	12	-	-	-	73
	Percentage	28.7%	17.8%	36.9%					83.5%	16.4%				100%

Source	Description
f	Letter indicating a framework. W - Eclipse Workbench, S - Apache Struts, A - Java Applet, and E - Sun EJB3.0
Df	Features from API documentation, such as Javadoc or other specifications, where f is the framework.
Sf	Features from schemas contained in the framework, where f is the framework.
Tf_i	Features from the framework's tutorials, where f is the framework and i is the identification number for the tutorial.
Af_i	Features 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	Implied features. For example, every class has a name, so a feature name is implied.
E	Features from expert knowledge.
EA	Features example applications that use the framework.
X	Extra features added by FSML developer. These features were related to a view filtering capability of the FSM editor.
F*	All features of the FSML.

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 and supported artifact types are listed in Table 1.

Vertical scope refers to the depth of the feature models. The deeper the models, the more detailed and fine-grained features become. 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. In particular, adding support for forward and round-trip engineering to an existing FSML that only supported reverse engineering will sometimes require deeper feature decomposition to model alternative implementation variants. Planning the horizontal scope is the main goal of I4; the vertical scope will more clearly emerge in later stages as the use cases are implemented.

Languages targeting API understanding are likely to cover the entire API. Their horizontal scope may emphasize the most used areas and perhaps also the more tricky ones. The horizontal 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 workbench part interactions for WPI. Languages supporting integrity checking, e.g., Struts FSML, are quite selective in terms of their horizontal scope. Model analysis involves adding higher level concepts for representing the results of the model analysis. Also, the extracted model is typically scoped toward 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.

5.3 Elaboration

E1: Identify framework-provided concepts in the scope (cf. Box 1).

? Which concepts should be included in the current iteration?

! Use the identified sources of knowledge to find relevant concepts.

API documentation, tutorials, and articles. These sources provide concepts and features representing the intended API use as envisioned by the framework developers.

Experience and best practices. These sources provide concepts and features on how expert developers use the framework. Note that experts often contribute articles and tutorials.

Sample applications. Analysis of sample applications, both supplied with the framework and real-life applications, may provide information about not only the typical API uses, but also the unusual ones, which may not be available from other sources.

Metamodels and XML schemas. These sources provide concepts and features that are made explicit by framework developers. Note that various configuration dialogs and wizards supplied together with the framework can be based on these metamodels or schemas.

E1: Identify framework-provided concepts in the scope

The following fragments of the Java Applet Tutorial [33] are used as a running example. We identify a single concept: *Applet*.
"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 `getParameter` method."

"You should also implement the `getParameterInfo` method so that it returns information about your applet's parameters."

Box 1: An example for activity E1

E2: Determine kinds of concepts in scope

The concept `Applet` is a component-oriented concept because it is a Java class and it does not reference other concepts.

The concept `ViewPart` in WPI FSML represents a graphical view in Eclipse. It is a component-oriented concept because it is a Java class and it does not reference other concepts.

The concept `AdapterRequestor` in WPI FSML is a connector-oriented concept. It relates a part that requests an adapter with multiple `AdapterProviders`. The concept corresponds to an adapter request method call.

The concept `BusinessInterface` in EJB FSML is a port-oriented concept. The concept 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 using Java annotations (*@Local* or *@Remote*) or through the XML deployment descriptor.

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 a separate 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 separate concepts; however, they could also be modeled as subfeatures of parts. In EJB FSML, *explicit* business interfaces are modeled as separate concepts and then are referenced, whereas *derived* business interfaces are subfeatures of other concepts.

Box 2: Examples for activity E2

E2: Determine kinds of concepts in scope (cf. Box 2).
? What is the nature of the concepts?
! Classify concepts as component-oriented, connector-oriented, or port-oriented.

Component-oriented concepts are those that do not relate other concepts with each other; rather, they aggregate other concepts and features. Component-oriented concepts usually correspond to code components, such as Java classes or XML documents.

Connector-oriented concepts model connections among other concepts using references. Connector-oriented concepts can correspond to code components (e.g., classes implementing listeners), but they can also correspond to code patterns scattered across the components related by the connector, such as scattered method calls or XML declarations.

Port-oriented concepts interface between the components and connectors, i.e., they enable connecting a connector to a component. They often correspond to Java interfaces.

E3: Define the overall structure of the feature model (cf. Box 3).
? How should the feature model be organized?

A feature model of an FSML is a hierarchy of features. We represent the hierarchy using indentation, i.e., subfeatures are indented further to the right.

E3.1: Define a model root. A model root is the container of all concepts and features and it always corresponds to the entire application.

E3.2: Introduce features for grouping related concepts/features. A feature model can be divided into logical parts that group related concepts or features. This step may require the introduction of new concepts or features whose only purpose is to group its children. The grouped elements could be related conceptually or based on the source code artifact they represent.

E3.3: Characterize concepts as base or derived concepts. Instances of derived concepts correspond to the same code patterns as instances of base concepts do. Code patterns that instances of base concepts correspond to are specified directly in the metamodel using mapping definitions. In contrast, code patterns that instances of derived concepts correspond to are specified using a *base-concept reference*, which can point to an instance of the base concept (cf. Box 3). The base-concept reference mechanism is useful in at least three ways. First, it helps reduce redundancy in the metamodel because essential features of base concepts need not be copied to the derived concepts. Second, it allows for specifying derived concepts as standalone concepts rather than subfeatures of their base concepts. Third, it supports composition of multiple FSMLs by allowing concepts in one language to be derived from concepts of another language.
! For derived concepts, create a base-concept reference subfeature typed with the base concept.

5.4 Construction

In this phase, the feature model is refined into a complete metamodel and any additional required artifacts, such as queries, transformations, and test suites, are created.

C1: Decompose the concepts into features.
? How should the concepts be decomposed into features?

C1.1: Choose appropriate level of abstraction and granularity (cf. Box 4).
? Which features are important with respect to the value proposition? Which features can be abstracted away?
! Use the criteria below to decide which features are potentially useful to include.

E3: Define the overall structure of the feature model

```
AppletModel
  Applet
    overridesLifecycleMethods
```

The concept `Applet` is a child of the model root `AppletModel`. The feature `overridesLifecycleMethods` is used for grouping features corresponding to the life cycle methods.

```
EJBApplication
  InformationFromAnnotations
  InformationFromDeploymentDescriptor
```

In EJB FSML, the feature `InformationFromAnnotations` is used for grouping subfeatures related to Java annotations and the feature `InformationFromDeploymentDescriptor` is used for grouping subfeatures related to the XML deployment descriptor.

```
WorkbenchPartInteractions
  Part
  ViewPart -|> Part
  EditorPart -|> Part
  AdapterRequestor
    requestor (Part) <baseConcept>
```

The concept `Part` is a base concept and the `AdapterRequestor` is a derived concept. The feature `requestor` is a base-concept reference of the concept `AdapterRequestor` and it specifies that each instance of the concept `AdapterRequestor` will correspond to the same code pattern that the referenced base-concept instance corresponds to, which is an instance of the concept `Part` in this case. Note that instances of both `ViewPart` and `EditorPart` concepts can be values of the `requestor` reference. Alternative designs (without the base-concept reference mechanism) are as follows:

```
Part                  Part
  AdapterRequestor      ViewPart -|> Part
ViewPart -|> Part       EditorPart -|> Part
EditorPart -|> Part     ViewAdapterRequestor -|> ViewPart
                        EditorAdapterRequestor -|> EditorPart
```

In the first case, the `AdapterRequestor` is not a standalone concept and in the second case, inheritance is used to make sure that the adapter requestor concepts are views and editors. Both alternative designs are suboptimal and do not reflect modeler's intention. Additionally, the base-concept reference mechanism enables seamless FSML composition. For example, the concept `AdapterRequestor` could be defined in a different FSML.

Box 3: Examples for activity E3

Features corresponding to implementation steps. Any API usage ultimately involves some implementation steps, such as subclassing a framework class, implementing a framework interface, implementing callback methods, invoking a framework service, placing a Java annotation, creating framework-stipulated XML declarations, or setting XML attribute values. Every FSML will include some amount of these features.

Frequently used features. These features are likely to increase the usefulness of the language.

Composite features. These features are defined in terms of other features. Higher-level abstractions are defined as composite features over features representing individual implementation steps. Such higher-level features usually add significant value to the language.

Features related to complex API constraints or dependencies. Including these features in the language will enable checking these constraints in FSMs.

Features related to protocols. Ordering of framework-related calls is typically enforced by frameworks (callback protocols); however, application programmers are sometimes required to follow protocols related to the life cycle of framework components. These protocols incur constraints such as that certain framework services must be invoked in conjunction and/or in a certain order. Such constraints are easy to violate in the code and therefore including features that represent them in the language should be considered.

Features with variability. Features that have significantly

C1.1: Choose appropriate level of abstraction and granularity

```
AppletModel
  Applet
    extendsApplet
    overridesLifecycleMethods
      init
      destroy
    registersMouseListener
      deregisters
    parameter
      name
    Thread
      InitializesThread
       !<1-1>
          initializesThreadWithRunnable
          initializesWithThreadSubclass
    providesParameterInfo
    providesInfoForParameters
```

Implementation steps. Most of the features correspond to implementation steps. For example, in order to create an instance of the concept `Applet`, the developer must create a Java class which extends the framework class `Applet`, override life cycle methods (e.g., `init` or `destroy`), register and deregister mouse listeners by adding calls to the appropriate methods, or retrieve values of parameters by adding a method call and specifying the name of the parameter (cf. Box 1).

Frequently used features. Examples of features that almost every applet has are the features `init` and `parameter`.

Features involved in API constraints. The features `parameter` and `providesParameterInfo` are involved in a constraint `providesInfoForParameters`. The constraint asserts that if at least one parameter is used, the applet should provide information about the parameters.

Features with variability. The feature `Thread` is an example of a feature with two functionally equivalent implementation variants. In Java, a thread can be initialized in two ways: by instantiating the `Thread` class or subclassing the `Thread` class. The choice is modeled using an *exclusive-or* feature group `!<1-1>`.

Scattered features. In our exemplar FSMLs most composite features are scattered features, for example the feature `Thread`.

```
StrutsApplication
  StrutsConfig
    ActionDecl
      path
    FormDecl
    ForwardDecl
```

Composite features. The feature `StrutsConfig` is a composite feature because it groups all features related to the action, form, and forward declarations in the `struts-config.xml` file.

```
WorkbenchPartInteractions
  SelectionListener
    registersAs
     !<1-3>
        globalSelectionListener
          deregisters
            deregistersSameObject
              registersBeforeDeregisters
        globalPostSelectionListener
        specificSelectionListener
```

Features related to life cycle protocol. In our exemplar FSMLs, features related to listeners have a life cycle because the listener has to be first registered and later deregistered, when no longer needed. The feature `globalSelectionListener` corresponds to the registration method call and the feature `deregisters` corresponds to the deregistration method call. Furthermore, the features `deregistersSameObject` and `registersBeforeDeregisters`, specify that the same object must be used in both method calls and that the registration must occur before deregistration, respectively.

Features with variability. The feature `registersAs` is an example of a feature with three alternative functional variants. A selection listener can be registered as a global selection listener, a global post selection listener, or a specific selection listener. Registering a listener in different ways achieves different effects for the same feature. The choice is modeled using an *inclusive-or* feature group `!<1-3>` instead of an exclusive-or one because a listener of each kind can be registered simultaneously.

Box 4: Examples for step C1.1

different *functional* or *implementation variants* are likely candidates for inclusion in the language. Functional variants achieve different functional effects, whereas implementation variants are different ways of implementing the same effect.

Scattered features. These features correspond to code patterns scattered across the completion code. They can correspond to the scattered patterns directly or through their subfeatures. The key value of supporting scattered features is that they can be localized in the model and ease the access to and comprehension of the corresponding code patterns that are scattered across the code. Effectively, an FSM can represent alternative code decompositions.

C1.2: Decide on feature nesting (cf. Box 5).

? Which features should be parent features and which should be subfeatures?

! In general, feature nesting may follow partonomies, taxonomies, and other dependencies [39]. Since features represent code patterns, a subfeature relationship between a parent and a child feature may correspond to *part-of*, *is-a*, or other dependencies between the represented code patterns. Use the criteria below to decide on feature nesting.

Code pattern nesting. Feature nesting may mirror syntactic nesting of the code patterns that the features correspond to.

Code pattern subtyping. Feature nesting may mirror subtype relationship between the code patterns that the features correspond to.

Other dependencies. Feature nesting may mirror other dependencies, such as declaration-use, control-flow, or data-flow dependencies, between the code patterns that the features correspond to.

C1.2: Decide on feature nesting

Code pattern nesting. The features corresponding to fields and methods are likely children of features corresponding to classes (cf. Box 4, the features `Thread` and `init` are children of `Applet`). Also, features corresponding to XML attributes are children of features corresponding to XML elements (cf. Box 4, the feature `path` is a child of `ActionDecl`).

```
AppletModel
  Applet
    extendsApplet
      extendsJApplet
    Thread
       ...
          initializesWithThreadSubclass
            overridesRun
```

Code pattern subtyping. Features corresponding to being assignable to more specific types (e.g., `extendsJApplet`) are children of features corresponding to being assignable to more general types (e.g., `extendsApplet`) because of the semantics of type inheritance (extending `JApplet` implies extending `Applet`).

Other dependencies. The feature `overridesRun` specifying that a subclass of the class `Thread` should override the method `void run()` only applies if the variant with a subclass is used (the feature `initializesWithThreadSubclass`).

Box 5: Examples for step C1.2

C1.3: Specify cardinality constraints (cf. Boxes 6, 7).

? What should be the valid number of instances of a feature in every feature configuration? Cardinality constraints of which features and feature groups are essential for an instance of their parent feature to exist?

! Choose between the kinds of cardinality listed below.

Feature cardinality is an interval $[m..n]$, $m \geq 0 \wedge (n \geq m \vee n = *)$, specifying how many instances of a given feature (at least m and at most n) should be present as children of that feature's parent in a feature configuration. In general, feature instances correspond to API-stipulated code patterns in the completion code and the cardinality of a feature depends on the possible number of patterns that can be matched.

Feature group cardinality is an interval $<m\text{-}n>, 0 \leq m \leq n \leq k$ specifying how many instances of features in a feature group of size k should be present as children of that group's parent feature in a feature configuration. Feature groups represent a choice over a number of grouped features. Feature groups with cardinality <1-1> model an exclusive-or choice, that is, that exactly one instance of one of the grouped features should be present. Feature groups with cardinality $<1\text{-}k>$, where k is the number of grouped features, model an inclusive-or choice, that is, that at least one instance should be present. A feature group with cardinality $<0\text{-}k>$ is equivalent to k optional features.

The semantics of a feature model is a set of legal feature configurations, that is, configurations in which all constraints (including cardinality constraints) are satisfied. We introduce the notion of essential constraints to define a superset of the legal feature configurations in which the essential constraints are never violated but in which other constraints can be violated. Essentiality can be thought of as a modifier on feature and group cardinality that specifies that both lower and upper bounds of the cardinality must not be violated. Knowing that certain constraints will never be violated in any configuration from that superset allows us to detect and reason about incorrect concept instances. Essentiality of additional constraints can be expressed by modeling them as features (cf. C1.4) whose cardinality is essential.

Essential features. These features correspond to patterns that are the minimum requirements to identify a concept instance. Without its essential features an instance cannot be considered an instance of a particular concept. Essential feature is a short form for *essentially mandatory feature*. We mark cardinality of essential features using the exclamation point (!), that is, ![1..1].

Essential feature groups. Often, several different essential features are possible and more than one may or may not be allowed to be present at the same time. Such situations can be modeled using *essential feature groups* !<m-n>.

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].

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 among them, such as that certain patterns are alternative

C1.3: Specify cardinality constraints

```
AppletModel
  [0..*] Applet
   ![1..1] extendsApplet
      [0..1] extendsJApplet
    [0..*] Thread
     ![1..1] typedThread
      [1..1] InitializesThread
       !<1-1>
          [0..1] initializesThreadWithRunnable
          [0..1] initializesWithThreadSubclass
      [1..1] nullifiesThread
    [1..1] providesInfoForParameters
EJBProject
  [0..1] InformationFromAnnotations
    [0..*] EJBClass
    [0..*] SessionBean -|> EJBClass
      [0..1] remoteInterfaceSpecification
       ![1..*] remoteInterfaces (RemoteLocalInterface)
          [1..1] interfaceName
    [0..*] StatelessEJB -|> SessionBean
     ![1..1] statelessAnnotation
  [0..1] InformationFromDeploymentDescriptor
    [0..*] DDStatefulEJB -|> DDSessionBean
     ![1..1] sessionType
       ![1..1] isStatefulSessionType
```

Essential features. Essential features of component-oriented concepts may correspond to being assignable to a certain type (`extendsApplet` or `typedThread`), being annotated with a Java annotation of a certain type (`statelessAnnotation`), or being declared as a component in an XML configuration file (`isStatefulSessionType`). Essential features of connector-oriented concepts may correspond to method calls that attach the connector to a component (cf. Box 7, `requestsAdapter`). Finally, an essential feature of port-oriented concepts may correspond to explicitly naming the port in a Java annotation of the component (`remoteInterfaceSpecification` and `interfaceName`).

If an instance of the essential feature `extendsApplet` was missing, a given class would not be considered an instance of the concept `Applet`. Similarly, a class will only be considered an instance of a stateless EJB in the annotation section of the language (that is, under the feature `InformationFromAnnotations`) if the class is annotated with a stateless annotation (the feature `statelessAnnotation`).

Essential feature groups. The feature group `!<1-1>` specifies that an instance of `InitializesThread` cannot exist without an instance of at least one of the grouped features.

Mandatory features. For example, the features `initializesThread` and `nullifiesThread` are required for the correct implementation of a thread but they are non-essential since only the feature `typedThread` is required to determine that a given field is a thread.

Often, mandatory features are used to model API constraints, so that a missing instance of a mandatory feature indicates constraint violation. For example, the feature `providesInfoForParameters` corresponds to an API constraint.

Optional features. For example, the feature `extendsJApplet` indicates that the developer can optionally extend another API class. Examples of optional grouped features are `initializesThreadWithRunnable` and `initializesWithThreadSubclass`.

Box 6: Examples for step C1.3

(exclusive-or).

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 $m \geq 1 \wedge (n > m \vee n = *)$. Essential multiple is a short form for *essentially mandatory and multiple*. Usually, concepts have cardinality [0..*] since there may be no or many concept instances implemented in an application.

Prohibited features. These features correspond to patterns that should not be present in the code. Prohibited features have cardinality [0..0]. Prohibited features may represent undesirable situations, such as API misuses,

C1.3: Specify cardinality constraints

```
WorkbenchPartInteractions
  [0..*] Part
  [0..*] EditorPart -|> Part
    [0..0] extendsMultiPageEditor
  [0..*] SelectionListener
     ...
        [0..*] globalSelectionListener
          [1..*] deregisters
  [0..*] AdapterRequestor
   ![1..1] requestor (Part) <baseConcept>
   ![1..*] requestsAdapter
```

Multiple features. The feature `globalSelectionListener` corresponds to the registration method calls and it is an optional multiple feature. In the feature configuration, each instance of the feature will correspond to a single method call. The feature `deregisters` is a mandatory multiple feature because for every registration method call there should be at least one deregistration call. The feature `requestsAdapter` 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. For example, the feature `extendsMultiPageEditor` corresponds to an editor extending a deprecated API class `MultiPageEditor`.

Box 7: Examples for step C1.3 (cont'd)

code smells, or uses of deprecated API elements. *Essentially prohibited features* have the cardinality ![0..0].

C1.4: Specify additional constraints (cf. Box 8).

? Which additional constraints are not captured by the feature hierarchy?

! Model additional constraints as features with attached model queries.

Not all constraints can be expressed by the feature hierarchy or the cardinality constraints alone, e.g., global constraints involving features from different parts of the hierarchy. Such additional constraints can be modeled as features, each with an attached model query. These features are usually mandatory, meaning that if the attached query evaluates to false or to an empty set, the modeled constraint is violated.

C1.4: Specify additional constraints

```
StrutsApplication
  [1..1] StrutsConfig
    [0..*] ActionDecl
      [0..1] type (String)
      [1..1] actionImpl (ActionImpl) <where attribute:
                    qualifiedName equalsTo: ../type>
  [0..*] ActionImpl
    [1..1] qualifiedName (String)
```

The feature `actionImpl` models a referential-integrity constraint. The constraint implies that every correct action declaration (`ActionDecl`) must have an action implementation (`ActionImpl`). To model this, `actionImpl` has a reference attribute of type `ActionImpl`, that is, the feature represents a reference to an instance of `ActionImpl`. For each instance of `ActionDecl`, the attached model query retrieves all instances of `ActionImpl` whose `qualifiedName` has the same value as `type` of the `ActionDecl` instance. Finally, the cardinality of `actionImpl` specifies that there should be exactly one such instance of `ActionImpl`.

Box 8: An example for step C1.4

A special case of such global constraints are referential integrity constraints. These constraints model correspondences between concepts, such as "every instance of a given concept with some property has a corresponding instance of another concept with some other property." Such a constraint can be modeled by a feature with a reference attribute and an attached model query (cf. Box 8). The query retrieves the corresponding instances and the

feature's cardinality specifies the expected number of the corresponding instances.

C1.5: Define features with reference attributes to connect concepts (cf. Box 9).

? Is the concept related to other concepts? For example, connector-oriented concepts are related to the concepts they connect.

! Define features with reference attributes that can point to instances of other concepts. Use <baseConcept> annotation to define base-concept references (cf. E3.3) or use model queries to locate instances of the concepts.

C1.5: Define reference features

```
WorkbenchPartInteractions
 [0..*] Part
 [0..*] AdapterProvider
  ![1..1] provider (Part) <baseConcept>
  ![1..1] providesAdapter
    ![1..*] adapters (String)
 [0..*] AdapterRequestor
  ![1..1] requestor (Part) <baseConcept>
  ![1..*] requestsAdapter
     [1..1] adapter (String)
     [0..*] adapterProvider (AdapterProvider) <where
                  attribute: providesAdapter/adapters
                  contains: ../adapter>
```

`AdapterRequestor` is a connector-oriented concept. It references 1) a part through the `requestor` base concept reference and 2) multiple adapter providers through the `adapterProvider` feature. The constraint specifies that the values of that reference should be all instances of the concept `AdapterProvider` whose list of provided adapters contains the requested adapter.

Box 9: An example for step C1.5

C2: Create mapping definitions for the features (cf. Box 10).

? What code patterns should concept/feature instances correspond to?

! Create mapping definitions to specify the correspondence between concept/feature instances 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.

Table 3 presents the mapping types that are used in the examples in Box 10. The complete set of the 49 mapping types used in the exemplar FSMLs can be found elsewhere [5], [22].

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 (cf. Table 4) to control code pattern matching, and forward parameters are used only by code transformations (cf. Table 5) to control code pattern creation. Some parameters are optional; we present them in square brackets, and we show the default values for these parameters following the bar symbol (|). In general, the parameters that are present in Table 4 but missing in Table 3 are reverse parameters. Similarly, the parameters present in Table 5 but missing in Table 3 are forward parameters. Furthermore, parameters can be specified *statically*, i.e., by providing their literal value in mapping definitions, or *dynamically*, i.e., by having their value retrieved from other features during RE, FE, or RTE.

C2.2: Specify static parameter values in the mapping definition.

! Specify literal values for parameters that should be specified statically, such as the name of an interface that the feature should correspond to. Extend the mapping type with new parameters if needed.

C2.3: Specify features for the retrieval of dynamic parameter values.

! For each parameter to be specified dynamically, decide whether the value should be determined implicitly using the *context mechanism* (explained shortly) or explicitly using a path. Ensure that the appropriate parent features from which values will be retrieved using the context mechanism exist. Adjust the order of features to make sure that the mapping definitions of features from which values are retrieved are evaluated before the mapping definitions using these values are evaluated.

The context mechanism is a convenience mechanism that obviates the need to explicitly specify the path to a parent feature from which the value needs to be retrieved. The context mechanism will pick the closest parent feature instance that corresponds to a code pattern which is assignment-compatible with the type of the parameter. The code pattern is then assigned to the parameter. The context mechanism also allows for reuse of feature model fragments in different places in the hierarchy since explicit references to parent features do not have to be hardwired.

C2.4: Implement missing code queries and transformations.

! Implement new queries and transformations for newly defined mapping types. Extend existing queries and transformations if they need to better approximate their mapping types and handle additional code patterns.

Code queries are required for reverse and round-trip engineering. Code transformations are required for forward and round-trip engineering. Existing mapping types provide default implementations of code queries and transformations. Tables 4 and 5 present code queries and transformations for the mapping types in Table 3.

C3: Add key annotations (cf. Box 11).

? How should instances of concepts/features be identified?

Every concept/feature instance should be uniquely identifiable in an FSM. In the current infrastructure, instances are considered matching if their keys match. Keys are required for RTE because the correspondence between the instances from the descriptive, prescriptive, and last-reconciled models is established based on their keys. Keys are also needed for establishing traceability links during RE.

By default, a key for every concept/feature contains the concept's/feature's name; however, since there may be many instances of the same concept or feature in a single model, keys need to be enriched with additional information. This enrichment is specified using *key annotations* which are interpreted by the FSML infrastructure

C2: Create mapping definitions for the features

```
AppletModel <project>
  [0..*] Applet <class>
    [1..1] name (String) <fullyQualifiedName>
   ![1..1] extendsApplet <assignableTo: 'Applet'>
    [0..*] parameter <callsReceived: 'String getParameter(String)' location: 'void init()'>
      [0..*] name (String) <valueOfArg: 1>
    [0..1] providesParamInfo <methods: 'String[][] getParameterInfo()'>
    [1..1] providesInfoForParameters <constraint: ../parameter implies: ../providesParameterInfo>
```

Table 3 presents selected mapping types used in the example above. Values of some mapping type parameters were specified statically. Some parameters were left unspecified and their values will be retrieved using the context mechanism. For example, let's consider the mapping definition <assignableTo: 'Applet'> that uses the assignableTo: mapping type. The mapping type defines two parameters: *c* and *t*. In the mapping definition, only the value of the parameter *t* is set to 'Applet'. The parameter *c* is left unspecified leaving the determination of its value to the context mechanism during the FSML execution. The context mechanism will use the Java class that the closest parent feature corresponds to as the value of the parameter *c*. In our example, *c* will be set to the Java class that the concept instance Applet corresponds to. Analogously, the value of the parameter *mc* in the mapping definition <valueOfArg: 1> will be the method call that the feature parameter corresponds to. Target features for the retrieval of the code patterns for dynamic parameters can be always explicitly specified using path expressions, such as, ../parameter.

The semantics of the above metamodel is as follows. The model root AppletModel always has to correspond to the entire application (project). Each instance of the concept Applet corresponds to a Java class. Value of the feature name is the fully qualified name of the Java class. Instance of the feature extendsApplet will be present only if the Java class is assignable to the Applet class. Because the feature extendsApplet is an essential feature of the concept Applet, the instance of the concept cannot correspond to a class that is not assignable to the class Applet. Each instance of the feature parameter corresponds to a method call to getParameter received by an object of the class. Each value of the subfeature name of parameter corresponds to the value of the first argument of the method call. An instance of providesParamInfo corresponds to an implemented or overridden method getParameterInfo by the class.

An example of a forward parameter is location, used in the mapping definition for the feature parameter. The parameter specifies that the code transformation should create the method call in the body of the method init.

Box 10: Examples for activity C2

TABLE 3
Selected and simplified mapping types for structural and behavioural code patterns in Java

Structural Pattern Expression	**Structural Element(s) Matched**
project	matches a project
class	matches a Java class
c fullyQualifiedName	matches the fully qualified name of the class *c*
c assignableTo: *t*	matches if objects of the class *c* are assignable to the type *t*
c methods: *s*	matches methods with signature *s* that are implemented or overridden (but not inherited) by the class *c*
Behavioural Pattern Expression	**Run-time Event Pattern(s) 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 i^{th} argument of the method call *mc*

TABLE 4
Code queries for the mapping types from Table 3

Code Query Expression | Defaults	**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. *(The essential mode means executing the query for an essential feature. It is an optimization used to return only classes that satisfy the essential feature and thus avoid checking all classes in the system [5, s. 2.5.2].)*
c getFullyQualifiedName	A fully qualified name of the class *c*.
c getMethods: *s*	A 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*	A set of values of the i^{th} argument of the method call *mc* retrieved using interprocedural constant propagation limited in scope to the class that contains the method call.

when computing keys for instances.

! If a given concept/feature can only have a single instance in the entire model, do nothing. Otherwise, consider how the instances should be distinguished from each other by including keys or values of other instances.

If a value of a subfeature should be included in the key, put the annotation key on that subfeature. If the key of the parent instance should be included in the key, put the annotation parentKey on the concept/feature. If the concept/feature can be instantiated multiple times under a single parent instance, put the annotation indexKey to include the index of the instance in the sequence of all instances which are children of the same parent instance.

The annotation key should be used on features that contain distinguishing values, such as fully qualified name and field or method name. The annotations parentKey and indexKey can be used in conjunction, if necessary.

C4: Add support for new use cases (cf. Box 12).

? Which additional use cases can be added?

Usually, the evolution of mapping types, code queries, and transformations is driven by the need to achieve

TABLE 5
Code transformations for the mapping types from Table 3

Code Transformation Expression | 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 i^{th} argument of the method call *mc*. Adds a literal for a single value. For multiple values creates a variable and multiple assignments with values *v*.

C3: Add key annotations

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

By default, a key for every feature contains the feature's name. According to the key annotations, the key for `Applet` will also contain the applet's fully qualified name; for `extendsApplet`, it will contain the key of `Applet`; and for `parameter`, it will contain the key of `Applet` and an index of the instance in the list of parameters.

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. This position changes easily when the code is rearranged.

Box 11: An example for activity C3

the value proposition of a language. Often, however, it is possible to add support for new use cases with very little cost.

C4: Add support for new use cases

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 were available, adding support for round-trip engineering amounted to setting forward parameters in mapping definitions.

Box 12: An example for activity C4

C5: Build a test suite (cf. Box 13).
? What code patterns/feature configurations should the language support?
! Create completion code/feature configurations that are in scope of the language. Use the test suite to test the use cases.

The code should have both correct and incorrect patterns with respect to the mapping definitions. 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.

5.5 Transition

In Transition, the language is extensively analyzed, tested, and refined until the required quality and usefulness are achieved. No more new concepts and features are added (i.e., *feature freeze* in Figure 7); however, modifications to the metamodel required for fixing errors are allowed. 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. We group this phase's activities by the qualities they evaluate (cf. Section 4).

5.5.1 Syntactic quality

We omit details of syntactic quality evaluation (activity T1) since it can be performed by the tools (e.g., syntax checks).

5.5.2 Semantic quality

The semantic quality of an FSML is determined by the feasible validity and completeness of the API usages encoded in its metamodel with respect to the FSML scope.

T2: Perform semantics evaluation.
? Is the feature model satisfiable? Does the FSML completely and correctly model the relevant API usages? Is the implementation of code queries and 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?

T2.1: Check feature model satisfiability. Verify that the set of legal feature configurations is non-empty and that each feature is instantiated in some configuration satisfying the essential constraints. The latter requirement acknowledges the possibility of features used only to detect incorrect API usages (prohibited features). Correct additional constraints that may cause contradictions (cf. C1.4).

C5: Build a test suite

Test code

```
public class MyApplet extends Applet {
  public void init() {
    String color = getParameter("color");
    String paramName = "width";
    if (...)
      paramName = "height";
    ...
    String dim = getParameter(paramName);
  }
  public void start() { ... }
  public String[][] getParameterInfo() {
    return ...;
  }
}
```

Framework-specific model

```
[1] Applet
  [1] name ('MyApplet')
  [1] extendsApplet
  [1] lifecycleMethods
    [1] init
    [1] start
  [1] parameter
    [1] name ('color')
  [1] parameter
    [1] name ('width')
    [1] name ('height')
  [1] providesParamInfo
  [1] infoForParams
```

Generated code for the model

```
public class MyApplet extends Applet {
  public void init() {
    getParameter("color");
    String param0 = "width";
    param0 = "height";
    getParameter(param0);
  }
  public void start() {
  }
  public String[][] getParameterInfo(){
    return null;
  }
}
```

The above test code can be used for testing the reverse engineering use case of the Applet FSML. The FSM is a result of reverse engineering the test code. The model contains an instance of the concept `Applet` and instances of its features. The instances are created for the underlined code patterns matched by the code queries.

The framework-specific model can be used for testing the forward engineering use case. The code generated for the model is the result of the execution of code transformations for each feature instance.

Round-trip engineering use case can be tested by performing changes to the code or to the model and reconciling the differences. Round-trip engineering first reverse engineers the descriptive model from the current code and compares it with the prescriptive model. Automatic reconciliation of changes is guided by the result of the comparison and developer's decisions.

Box 13: Examples for activity C5

T2.2: Check relevant API usages **T2.3: Evaluate mapping definitions**

```
AppletModel <project>
  [0..*] Applet <class>
    [0..*] parameter <callsReceived: 'String Applet.getParameter(String)' location: 'void init()'>
      [0..*] name (String) <valueOfArg: 1>
    [0..*] Thread <field>
      [1..1] InitializesThread
       !<1-1>
        [0..1] initializesThreadWithRunnable <assignedNew: 'void Thread(Runnable)' position: 'after' loca-
                                                                                    tion: 'void init()'>
        [0..1] initializesWithThreadSubclass <assignedNew: initializer: true subtypeOf: 'Thread'>
    [0..*] singleTaskThread <callsTo: 'void Thread(Runnable)' statement: true>
```

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

For example, applet threads can also be 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 `<1-1>` and adding new features starting with `initializesWithThreadSubclass`.

We also noticed, 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 from the feature `singleTaskThread`. We also had to extend the mapping type `callsTo` by introducing a new parameter `statement` that specifies whether only method/constructor calls that are statements should be matched by the code query.

T2.4: Evaluate the implementation of code queries and transformations

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. The mapping definition of the feature `initializesThreadWithRunnable` uses the first variant, and the assignment will be generated at the end of the method `init`. The mapping definition of the feature `initializesWithThreadSubclass` uses the second variant by setting the value of the forward parameter `initializer` to `true`.

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 [40]. 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 [14].

The feature models still 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` 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..*].

Box 14: Examples for steps T2.2, T2.3, and T2.4

T2.2: Check relevant API usages (cf. Box 14). Create sample feature configurations and verify that all legal feature configurations represent correct API usages and that illegal feature configurations represent incorrect API usages (validity). Verify that every relevant API usage is represented in the feature model (feasible completeness). Use the API definition, sample applications, and best practices to identify missing API usages. Extend the feature models and mapping definitions as needed.

T2.3: Evaluate mapping definitions (cf. Box 14). Verify that the mapping definitions correctly represent the correspondence between feature instances and code patterns as required by the API usages in scope.

T2.4: Evaluate the implementation of code queries and transformations (cf. Box 14). Refine code queries and transformations to improve the semantic quality of the retrieved models (precision and recall) and the generated code (correctness and completeness). Adjust the feature

model to the refined queries and transformations. Verify that the queries and transformations correctly use all mapping type parameters. Verify that code patterns created by code transformations are matched by code queries for the same mapping types.

5.5.3 Pragmatic quality

The pragmatic quality of an FSML needs to be evaluated at two levels: the level of FSMs and the FSML-metamodel level. Thus, the evaluation involves determining the feasible comprehension, creation, and modification of FSMs and the FSML metamodel.

T3: Perform pragmatics evaluation.

? Is the model/metamodel easy to understand, create, and modify?

T3.1: Evaluate model/metamodel comprehension, creation, and modification (cf. Boxes 15, 16).

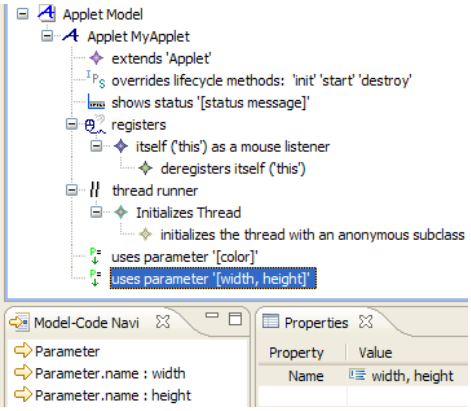
Evaluate closeness of mapping, consistency, error-proneness, hard mental operations, hidden dependencies, role-expressiveness, viscosity, and visibility dimensions for sample modeling tasks by asking questions similar to those included in the Cognitive Dimensions questionnaire [41], e.g., "What kind of things are more difficult to see or find? Do some kinds of mistake seem particularly common or easy to make?" Use readily understood feature names. Use appropriate concrete syntax, e.g., characteristic icons and visualization of relationships (designing concrete syntax is outside of the scope of the current method). Consider model filtering, transformation, and analysis to display different views of the model for different comprehension tasks.

For the metamodel, evaluate the decomposition of concepts into features and metamodel modularity (e.g., use of references and constraints). Detect redundancies and use references and feature inheritance where appropriate.

T3.2: Evaluate support for the required use cases (cf. Box 17). Verify that the language can be used to perform the required use cases. Use the test suite to execute the required tasks.

T3.1: Evaluate model/metamodel comprehension, creation, and modification

Screenshot: concrete syntax of an FSM expressed using Applet FSML.



Examples of cognitive dimensions evaluation for exemplar FSMLs.

Closeness of mapping. The main purpose of the EJB FSML is to present a run-time configuration view by merging information from the Java code and the XML deployment descriptors, which is very difficult to see both in the code and in an FSM reverse engineered using EJB FSML. Therefore, to improve the pragmatic quality of the retrieved EJB FSML models with respect to the closeness of mapping, we implemented a model transformation that applies inference rules to the extracted model and computes a run-time configuration of the EJB project and thus making the run-time configuration explicit [22].

Consistency. Different kinds of listeners in Applet FSML are similar in structure. Learning one kind of listener (e.g., mouse listener) allows the user to easily "guess" how to use other kinds of listeners (e.g., key listener).

Terseness. All FSMs are terse because they only contain information relevant to their value proposition and abstract away unrelated information from the code. Still, the models can be large making it difficult to locate needed information. However, due to their hierarchical nature, FSMs can be explored top-down by expanding certain branches of the tree while other branches remain collapsed. FSMs also do not contain all necessary information needed for code creation because some information is encoded in the mapping definitions as defaults.

Additionally, not all features are presented as a feature hierarchy. Features with attributes are presented as *properties* shown in a separate view (cf. screenshot, lower right) of the features shown in the hierarchy. Descriptive labels of hierarchy features also contain values of the property features that are not visible in the hierarchy.

Error proneness. Our model editors are structured, which reduces error proneness. It is still, however, possible to construct an incorrect model. An automatic and generic model validation implemented in the infrastructure can be used to find errors in the model, e.g. violations of additional constraints.

Hard mental operations. In FSMs in Struts FSML, it was difficult to find an action implementation (in Java) for a given action declaration (in XML). Therefore, a referential integrity constraint was introduced that allows a developer to quickly navigate to the implementation or determining that it is yet missing.

Box 15: Examples for step T3.1

5.5.4 Organizational quality

The organizational quality of a FSM or a FSML is the extent to which the model or language fulfills the goals of the users in an organization.

T4: Perform organizational evaluation (cf. Box 18).

? Is the language fulfilling its purpose and providing the value as expected? 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?

! Perform user studies to assess whether the language delivers its value proposition.

6 Method Justification

We justify the presented method by tracing its origins to the base approaches, which are well-established, and to the actual experience of developing the four exemplar FSMLs (cf. Figure 6). Our discussion accounts for the origin of each step and also gives insight on how the steps were scheduled and executed during the actual FSML development. As expected, the reality of that development was more complex and exploratory than the method communicates or needs to communicate [42]; however, this section gives a more complete picture of the actual development and it provides ideas of how the method may be specialized when applied.

6.1 Division of the lifecycle into four phases

The four lifecycle phases, Inception, Elaboration, Construction, and Transition, were borrowed from the Unified Process. Each FSML had a clear Inception phase,

T3.1: Evaluate model/metamodel comprehension, creation, and modification

Hidden dependencies. The number of features representing constraints that crosscut the feature hierarchy and referential integrity constrains is very low relatively to the total number of features (WPI FSML: 5/71, Struts FSML: 5/47, Applet FSML: 1/75, EJB FSML: 4/73) and therefore number of hidden dependencies is low.

Pragmatic quality of FSMs with respect to hidden dependencies is also improved by offering traceability links (cf. screenshot, lower left part) between the model and the code that allow the developer to immediately jump to code patterns that implement a given feature.

Role expressiveness. To improve role expressiveness (as compared to the generic feature configuration from Box 13), we developed simple concrete syntaxes that utilize descriptive labels for features and icons (cf. screenshot).

Viscosity. Removing or modifying features involved in constraints and referenced from other features may trigger constraint violations in remote parts of the model. In out FSM editors, model validator can be used to asses the impact of changes. Later, the changes can also be undone.

Since code pattern removal is not implemented, removal of a feature in the model requires manual removal of a corresponding code pattern.

Visibility. A model expressed using WPI FSML shows both local and external workbench parts. External parts are those visible in the class path of the project and they cannot be modified. Therefore, to improve visibility when working with local parts, we implemented simple model filtering that hides all external parts.

Box 16: Examples for step T3.1 (cont'd)

T3.2: Evaluate support for the required use cases

We performed extensive evaluation of reverse engineering [14] and tested forward and round-trip engineering [5, ch. 5, 6]. We evaluated the precision and recall of code queries by reverse engineering a large set of applications using WPI, Struts, and Applet FSMLs, and manually identified false positives and false negatives of code queries for behavioural Java code patterns (code queries for structural code patterns have 100% precision and recall). The results showed weighted average precision of 100% and recall of 96% for the behavioural code queries.

Tests of forward engineering using WPI and Applet FSMLs showed that the code created using code transformations is correct with respect to the respective APIs. We verified that automatically by reverse engineering the generated code and comparing the original and the extracted models, which were equivalent modulo feature ordering and inherited behaviour from superclasses.

Tests of round-trip engineering (RTE) using WPI and Applet FSMLs showed that 1) the code created in multiple iterations using RTE was equivalent to the code created using forward engineering; and 2) the model created in multiple iterations using RTE was equivalent to the model created using reverse engineering.

All these evaluations also (indirectly) confirmed the correctness of reverse, forward, and round-trip engineering algorithms and infrastructure support.

Finally, Cheema tested migration by semi-automatically migrating a real-world Struts-based application with 20 KLOC of Java to JSF [25]. The original code was first reverse engineered using the first version of the Struts FSML. Next, a specialized migration code generator was used to produce JSF compliant completion code. The code generator was guided by the model and it was capable of migrating not only the completion code, but also the bodies of actions. The migration did not include UI code.

Box 17: Examples for step T3.2

in which the goals and the scope of the language were defined. All goals of WPI and Applet were set at the beginning of the development; the support for code creation and evolution were only added later; however, the goals for Struts and EJB FSMLs shifted or were expanded during development (cf. Table 1). Thus, the Inception phase was revisited for Struts and EJB FSMLs in later iterations.

The evaluation of the languages, particularly testing activities, were done continuously after each change to the metamodel or the infrastructure; however, there were a few periods of extensive evaluation, such as the study of RE or evaluating EJB antipattern detection on open-source applications. Major parts of these evaluation

T4: Perform organizational evaluation

None of the exemplar FSMLs has been released to the public yet. Evaluation methods such as *controlled experiment*, *action research*, or *survey* can be used to evaluate the organizational quality of FSMLs.

Organizational quality can also be assessed using indirect measures, which are likely less expensive to obtain than performing user studies. For example, to evaluate the claim that framework-specific models modularize each concepts of interest as a single feature hierarchy and the corresponding code patterns are usually scattered over the completion code, we computed two metrics: concern diffusion over components (CDC) and concern diffusion over operations (CDO) [5, ch. 7], where each concern aggregates all features related to the same Java class, one implementing a view, an editor, an applet, or an action in our case. We computed the metrics on a large set of example applications and results showed that, indeed, the code patterns that implement the concepts are highly scattered. 70% of Eclipse editors and views are implemented in two or more components and 55% of them in three or more components, where components are individual classes and XML documents. 60% of Struts actions are implemented in two or more components. Only 10% of Java applets are implemented in two or more components; however, the Applet FSML does not consider applet declarations in HTML files, which adds another component. Finally, at least 55% of all concerns are implemented in two or more operations. These indirect measures indicate that FSMLs can potentially reduce the effort of code comprehension by providing traceability to the scattered code patterns because framework-specific models present related features in a single hierarchy and thus provide an alternative to the way the code is decomposed.

Box 18: Examples for activity T4

periods can be viewed as Transition.

The distinction between Elaboration and Construction was more subtle, but it reflected the fact that some initial construction of the feature models took place before creating any mapping definitions and detailed features. We classified the creation of these initial feature models as Elaboration. These initial feature models were fairly shallow, at most one or two levels below the model root. Some grouping features were created at level two during elaboration; additional ones were created later in Construction as needed. The feature models were refined and their mapping definitions were created later, and we classified these activities as Construction.

6.2 Use-case-driven development

Support for use cases was added incrementally, as prescribed by the method. Code understanding (U) was the primary use case for each exemplar FSML except for the Struts FSML, which was initially built to support migration (M). Code creation (C) and code evolution (E) were considered for Applet and WPI FSMLs from the beginning, but the actual implementation of the use cases was sequenced. The FSML metamodel was first built for U and tested. Then it was refined for C and tested. Then it was further refined for E and tested. The necessary refinements were not extensive and they were mostly additive, such as providing additional parameters for code transformations in mapping definitions, or adding detailed features representing implementations variants. When transitioning from C to E, some keys had to be verified and adjusted in order to achieve the desired model matching. Furthermore, some code transformations needed adjustment and, consequently, additional parameters had to be provided in mapping definitions. The fact that these refinements were not extensive and mostly additive is not surprising since the mapping definitions define the semantics of the features

regardless whether they are realized by queries for RE or transformations for FE or RTE. For Struts, the initial language designed for M was first redesigned to support U because the focus of the language shifted. The support for referential integrity checking (RIC) was subsequently added by mostly creating new features with reference attributes and appropriate constraints. At last, support for C was added to Struts opportunistically, mostly because all the mapping types used by the language were already supported by existing transformations. The EJB FSML development started with U, followed by RIC (similar to Struts) and API constraint checking (ACC). The support for model analysis (MA) was added last. Again changes to the EJB FSML through these iterations were mostly additive.

6.3 Tracing activities and steps to the development of the exemplar FSMLs

All activities and steps presented in the previous section, except step T3.1 and activity T4, were performed during the development of the exemplar FSMLs as illustrated by the examples in Boxes 1–17. Step T3.1 and activity T4, which require user studies, were added to the method for completeness as suggested by the model quality framework. All of the performed steps were executed for every exemplar FSML; however, the step execution varied among the FSMLs. For example, E2 distinguishes among component-oriented, connector-oriented, and port-oriented concepts. All exemplar FSMLs have component-oriented concepts; only WPI, EJB, and Struts FSMLs have connector-oriented concepts; and only EJB FSML has port-oriented concepts. Furthermore, steps related to queries and transformations, i.e., C2.4 and T2.4, would be executed differently depending on whether the given FSML needed transformations to support forward or round-trip engineering or not. In particular, all exemplar FSMLs support forward engineering except the EJB FSML, which did not require this capability to achieve its value proposition.

6.4 Tracing activities and steps to the base approaches

All of the steps in Inception and Elaboration can be seen as specializations of the generic steps given in the Feature Modeling Process [17, ch. 4], which is an extension of FODA [7]. Some steps in Construction, such as defining feature nesting (C1.2), specifying cardinality constraints (C1.3), and specifying additional constraints (C1.4), are specializations of generic feature modeling steps to the FSML context; however, the steps in activities C2-C5 are specific to FSMLs and do not have their counterparts in FODA. All of the steps in Transition are classified according to the quality criteria derived from the model quality framework and the Cognitive Dimensions discussed in Section 4.

6.5 Iterative development and scheduling of activities and steps

The FSMLs were developed iteratively. All of them except EJB were developed concurrently with the infrastructure; thus, the actual development process was more complex and exploratory than what the method suggests.

We can distinguish five major periods in this development: 1) development of the first WPI FSML prototype and later Struts FSML prototype with code reuse from WPI; 2) development of Applet FSML with a fully declarative metamodel that was driving the development of the generic FSML infrastructure, followed by the migration of WPI and Struts FSMLs to the infrastructure; 3) the first RE evaluation for Applet, WPI, and Struts FSMLs [40]; 4) development of the EJB FSML and the second RE evaluation for Applet, WPI, and Struts FSMLs [14]; 5) the FE & RTE evaluation for Applet and WPI FSMLs [5].

Each of these periods involved multiple iterations. Iterations were not timeboxed: iterations involving infrastructure construction and implementation of code queries and transformations were longer; iterations involving mostly metamodel construction were shorter. The evaluation periods included both Construction and Transition phases since the FSMLs were refined in the course of the evaluations.

The development of the EJB FSML resembled closely the process prescribed by the method since most of the required mapping types (including the queries and transformations) and the infrastructure were already present. The EJB FSML development had five iterations: 1) development of an initial, basic version of the language, which included the development of missing code queries for Java 5 annotations; 2) adding support for XML deployment descriptors and extension of code queries for XML; 3) broadening the horizontal scope to cover more cases related to override rules; 4) development of model analysis and adding features required by the model analysis, and 5) development of EJB antipattern detection and adding required features.

In general, the iterations were not uniform as not every step, every activity, not even every phase of the method were performed in each iteration. Usually, adding new features was driving required extensions to the 1) algorithms and infrastructure; 2) mapping types; and 3) code queries and transformations. These extensions required continuous testing.

Closely related concepts were added together, other concepts were added and decomposed in depth one by one. Features were always added while keeping their mapping definitions in mind and specifying them later: we found it difficult to consider features without thinking about their mapping definitions at the same time. The reason was that mapping definitions often referred to other features through the context mechanism or explicit paths, so we had to understand how to place

and decompose new features in relation to the existing ones.

The evolution of the metamodel was continuous. There were a few significant refactorings of the metamodels that invalidated existing models; however, most changes were additive and existing models were automatically migrated. The main reasons for metamodel evolution were 1) adding new feature groups and new implementation variants; and 2) adjusting to new capabilities of code queries and transformations. In addition to continuous testing in Construction, the evaluation periods involved review of metamodel structure and extensive testing of the mappings, queries, and transformations.

6.6 Development effort

The effort required to develop a new FSML depends mostly on the availability of the required mapping types and their implementations. Assuming that all required mapping types are implemented and the developer is familiar both with the framework and the FSML method, a new FSML can be built in the order of days thanks to the declarative nature of FSML definition. Implementing new code queries and transformations and any necessary infrastructure extensions can be time consuming; however, Master's students without previous knowledge of the target framework and without experience in the FSML approach were capable of building an FSML within weeks, which included implementing missing mapping types. These students had access to the first author to ask questions about the approach and the infrastructure since a description of the method did not exist at that time.

7 Discussion and Future Work

In the spirit of the design science paradigm (cf. Section 2), we now turn to discuss the adherence of our research approach to the paradigm and the maturity and limitations of the individual elements of the FSML approach.

Hevner et al. proposed seven guidelines that should be followed when applying the design science (DS) paradigm [8]. We list these guidelines and briefly discuss to what extent our research adheres to them.

Design as artifact. *DS research must produce one or more artifacts in the form of a construct, a model, a method, or an instantiation.* Our research led to the new construct of FSMLs, four exemplar FSMLs as instantiations, and a FSML engineering method.

Problem relevance. *The produced artifacts must address an important and relevant problem with respect to a constituent community.* Difficulties of framework API usage represent a practical problem [43], [44], and its relevance is highlighted by a plethora of approaches designed to address it (cf. Section 8).

Design evaluation. *The utility and quality of the artifacts must be demonstrated through evaluation.* We established the criteria for evaluating FSMLs in Section 4. The exemplar FSMLs were evaluated as prescribed, modulo user studies. The FSML engineering method has not been evaluated. We return to the topic of evaluation in the following two subsections.

Research contributions. *DS research must yield new contributions related the artifacts, foundations, or methodologies.* We claim the exemplars and the FSMF as artifact-related contributions since they address the limitations of other approaches as argued in Section 8; however, experimental comparison is future work.

Research rigor. *DS requires the application of rigorous methods in both artifact construction and evaluation.* The FSMF has been implemented in the form of a generic infrastructure; the infrastructure has been extensively tested using the exemplars; and the exemplars have been carefully evaluated, e.g., using precision and recall. The engineering method is based on well-established approaches, which allowed us to systematically and thoroughly elicit the modeling patterns and strategies used to construct the exemplars. A formalization of the FSMF, however, is still required.

Design as a search process. *DS research is inherently iterative.* FSMLs, FSMF, and the method were constructed and evaluated iteratively. The results of the evaluations were used as feedback to improve the artifacts.

Communication of research. *DS research must be communicated to both technical and managerial audiences.* To address the needs of managerial audiences, we presented examples of use cases and value propositions and discussed development effort. The artifacts are also documented in sufficient detail to allow a technical audience to recreate them [5], [14].

In the following sections, we elaborate on the the maturity of the different elements of the FSML approach and their limitations and suggest future work.

7.1 Exemplar FSMLs

The exemplar FSMLs target four widely used Java frameworks. The FSMLs are declaratively specified and their metamodels are interpreted by the infrastructure. Reverse engineering was extensively evaluated for WPI, Struts, and Applet FSMLs [14]. Forward engineering and round-trip engineering were tested for WPI and Applet FSMLs [5]. Use cases U, RIC, ACC, MA, C, E, and M were tested by FSML developers, but the results were not published. The use case AU was performed by several members of our research group who learned the FSML approach.

The key limitations of the exemplar FSMLs are as follows.

Lack of user studies. The FSMLs need to be evaluated in user studies. Ultimately, their usefulness needs to be confirmed in practice. To this end, user communities for these or additional languages should be established. The effectiveness of the languages could then be studied using surveys, case studies, or action research.

Limited generality of the exemplars. The exemplar FSMLs target widely used Java frameworks with different characteristics; however, the exemplar FSMLs will only be representative for those API concepts of other frameworks that can be modeled using the same mapping types as the ones used in the exemplars. We will return to this issue shortly.

7.2 FSML engineering method

The presented FSML engineering method was constructed by specializing the base approaches to the FSML context and by generalizing from the experience of building the four exemplars. The method was first presented in the related Ph.D. thesis [5] and significantly revised and extended in this paper, most importantly by applying the model quality and Cognitive Dimensions frameworks.

The key limitations of the method are as follows.

Created by retrospection. Since we do not have a detailed record of each language's development history, we relied on retrospection and we might have missed some details. We addressed this threat by looking at the current [5] and past [40], [45] versions of the FSMLs.

Parallel infrastructure development. WPI, Struts, and Applet FSMLs were developed in parallel to the development of the FSML infrastructure; therefore, their development might not be representative. We addressed this threat by providing more detail on the actual development and justifying our method design in Section 6. Also, as the number of FSMLs built increases and the foundation matures, the number of changes to the FSML infrastructure required to support new requirements will likely decrease. We observed that effect in the development of the four exemplars. In particular, the development of the EJB FSML required much fewer infrastructure extensions than the previous exemplars.

Lack of user studies. The FSML engineering method needs to be tested by third-party users. To that end, we envision conducting an exploratory experiment in which subjects would apply the method to a well defined part of a framework's API. The goal of the experiment would be to understand the questions and issues the subjects faced during the method application and how these questions and issues impact the quality of the produced FSMLs.

7.3 FSMF

The Framework-Specific Modeling Foundation (FSMF) was described elsewhere [5]. The generic FSML infrastructure is the implementation of the foundation. The infrastructure implements the means used by the use cases and it supports the four exemplar FSMLs. The mapping types are implemented by code queries and transformations as pluggable mapping interpreters.

The key limitations of the FSMF are as follows.

Generality of the mapping types. The set of mapping types we developed can be used to model a great variety of API concepts, but it is certainly not exhaustive. Thus, the creation of new mapping types is considered as a part of the method. The set of mapping types that can be supported is limited by the ability to implement the corresponding code queries and transformations. In particular, implementing effective queries and transformations for highly dynamic code patterns, such as complex object structures typically required by graphical UI frameworks (e.g., Java Swing), may be difficult or infeasible. Code queries for such code patterns may require sophisticated static analyses, which would significantly increase the time of RE and RTE or reduce the semantic quality (i.e., precision and recall) of the retrieved models. Furthermore, the existing mapping types only support Java and XML; however, mapping types for other languages, such as C/C++ or Ruby, could be implemented.

Code pattern addition only. The current support for RTE is limited in the model-to-code direction to automatic code pattern addition only. Manual additions, modifications, and removals in the code are automatically reflected in the corresponding FSMs upon synchronization. We experimented with automatic commenting-out of the code that corresponds to the concept and feature instances removed from the related FSM; however, this capability needs to be further developed and studied. Similarly, automatic code modification, such as refactoring one implementation variant into another one to reflect the corresponding change in the related FSM, needs further investigation.

Fixed locations for code pattern addition. A challenge in automatic forward and round-trip engineering is the choice of the locations for code pattern addition, i.e., new code patterns are added to code locations predefined in mapping definitions, which may require application developers to move code patterns to suitable locations. Lee et al. proposed and prototyped a number of new FSML-based means [46] to address this challenge. These means can be invoked directly in the source code, but with an FSM hidden in the background or shown on the side. *Framework-specific content assist* can be used to add implementations of new concept or feature instances directly in the code; *framework-specific keyword programming* provides keyword-programming support [47] based on FSML feature names; *framework-specific content outline* presents an FSM of the currently opened source file; and *framework-specific error highlighting* marks API violations detected through an FSM directly in the source code. These new means required extensions to the FSML infrastructure, such as reverse navigability from code to FSM, specialized FSML metamodel interpretation, and extensions to code transformations.

Support for existing code. The main difference between FE and RTE, which both only add code patterns, is that in FE the code is created from scratch and in RTE the code is added to existing user's code. In FE, the FSML developer can easily predict the resulting code since code transformations will be executed for every feature. In RTE, some features may be already implemented in the

application and the added features must be seamlessly integrated. Currently, the same code transformations are used in both forward and round-trip engineering and therefore the quality of the code created in RTE may be lower than in FE. Also, the tests of RTE that we performed [5] did not involve third party applications.

Tensions between use cases. Currently, the method suggests that a single FSML metamodel can be used in many use cases, which is certainly the case with all exemplar FSMLs. In general, however, this approach might not always work because achieving high quality in one use case may conceivably reduce the quality of another use case or even render the other use case infeasible. The method prescribes incremental addition of support for new use cases which may require some restructuring of the metamodel. In our experience, this was never a problem since adding support for a new use case typically required only additions to the metamodel. We always started with the RE capability that only had features required for code comprehension (U). Adding FE, needed for code creation (C), typically required adding new subfeatures with information required by code transformations, such as field and method names. Adding RIC and ACC typically required adding features with reference attributes and features that represent constraints. Adding model analysis (MA) required creating a new metamodel for the view (i.e., the result of the analysis) and implementing an external model transformation. Adding migration (M) required implementing a specialized code transformation. Adding RTE after FE, needed for evolution (E), only required adding key annotations for instance matching, which is necessary for model comparison. Often, however, the required key annotations were already present since they were needed for establishing traceability links in RE. In general, different feature decompositions for different purposes could be provided as views as in the case of MA. Exploring effective ways of defining such views is future work.

Lack of formalization. Another limitation is the lack of formalization of the FSMF. The semantics of an FSML is specified as a semantic domain and a semantic mapping [48]. The semantic domain is a set of static and behavioral code patterns, and the semantic mapping is given by the mapping definitions. Although these concepts are currently not formalized, we have related mapping types to pointcut languages [40]. Thus, a formalization of the FSMF could build on the existing work on formal models of pointcut languages over static code patterns [49] and dynamic code patterns [50]. The complete semantic mapping for an FSML is given compositionally through the mapping definitions of its features. A formalization of this model is future work.

Furthermore, the use of code queries and transformations need to be formalized is in order 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 compositional definition of a code transformation in forward engineering. A formalization of round-trip engineering would provide insights into the properties of the approach, such as what exactly are the conditions of reconciliation and what are the limitations of instance identification (matching).

No guidance for concrete syntax. The method does not provide any guidance for building concrete syntaxes for FSMLs. Effective concrete syntax improves pragmatic quality and it is a prerequisite for any studies involving users. The creation of the concrete syntax could be driven by the concept kinds; e.g., FSMLs including both component-oriented or connector-oriented concepts may require a diagrammatic *lines-and-boxes* notation. The concrete syntax should ideally leverage results related to supporting feature model configuration, constraint propagation, and supporting different configuration interfaces, such as tree view and configuration wizards [51], [52], [53]. We developed simple concrete syntaxes for the exemplar FSMLs that utilize a generic graphical tree editor customized with icons and descriptive labels (cf. Box 15).

8 Related Work

8.1 Engineering of domain-specific languages (DSLs)

DSLs for framework APIs. The idea of designing modeling languages for frameworks is not new. In 1996, Roberts and Johnson proposed that language tools, such as visual builders, should be the last stage in the evolution of black-box frameworks [54]. The FSML approach provides a systematic method to building such languages declaratively.

In their 2005 survey on DSLs [55], Mernik et al. identify the following DSL development phases: decision, analysis, design, implementation, and deployment. The survey mentions 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, such as documentation, experience, and example code. Feature-oriented domain analysis (FODA) [7] 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 domain analysis. In our method, FODA is specialized and extended to support the complete definition and implementation of an FSML. Furthermore, our method gives specific guidelines on how the concepts and features are obtained, scoped, structured, mapped to code, and evaluated.

Bravenboer and Visser present MetaBorg [56], a method for embedding DSLs into general-purpose languages. A program expressed in an embedded DSL is

included inside a host program written in a general-purpose language. During the *assimilation* phase, the embedded DSL program is translated into the host language. Bravenboer and Visser demonstrate their approach by implementing a simple DSL on top of Swing, which is a Java framework for GUI programming. The assimilation of DSL programs produces the framework completion code. Code generation in MetaBorg is localized, i.e., it replaces the embedded DSL program with the completion code within the host program. In contrast, FSMs are external to the program using the API, and code generation for FSMs has global and incremental character with respect to the target program. 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 may be customized as needed thanks to round-trip engineering.

Domain-Specific Modeling Languages (DSMLs). DSMLs are specification-level DSLs for capturing requirements or designs. FSMLs can be viewed as a special kind of DSMLs for expressing the design of framework-based applications in relation to the framework API.

Most DSML development approaches (e.g., [57], [58], [59]) are based on metamodeling using some form of class models (e.g., [60], [61], [62]) or entity-relationship models [63]. The metamodeling notation for FSMLs is cardinality-based feature modeling [15], extended with reference attributes and inheritance. The expressiveness of the resulting notation is comparable with that of class modeling [16]. The main advantage of feature modeling are its explicit constructs for variability constraints such as feature groups. Furthermore, our metamodeling notation supports additional concepts such as essentiality and keys.

Kelly and Tolvanen present a process for designing DSMLs for 100% code generation [64]. 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 from a model never has to be manually modified. The development process consists of the following activities: selecting an appropriate domain, ensuring proper training and management support, defining the DSML, pilot project, and deployment. The authors advocate use scenarios to drive the development of DSMLs. They also provide guidelines on how to identify and define modeling concepts, design concrete syntax, and develop code generators using scripting and a template-based approach. The approach targets languages that are closer to system requirements and normally require significant amount of generated code that is not intended to be modified manually. In contrast, FSMLs target the use of frameworks and they usually are much closer to the respective framework APIs. They also support reverse engineering and round-trip engineering, which allows manual edits of the generated code. In contrast to DSMLs with 100% code generation, FSMLs target *model-supported engineering*, where models are optional since they can be retrieved from code at any time. In a sense, FSMLs follow a single-source principle, where all relevant information is stored in one place, which is source code of the application. FSMLs may also be of particular interest when the application domain of interest is not mature and stable enough to create a high-level DSML. FSMLs may have a lower adoption barrier, particularly in cases where an organization would already use frameworks for which FSMLs existed. Furthermore, due to the narrower scope of FSMLs, we are able to give more detailed guidelines and process for language development.

Quality of models and languages. Moody reviewed 50 different proposals to evaluating quality of conceptual models [65]. Some of these approaches are specific to particular notations such as UML or ER/EER. Only few of these approaches were empirically evaluated. The framework by Lindland et al. [9], which we use in our work (cf. Section 4), is one of the few general approaches that were evaluated. It is important to note that Lindland et al. also showed that other properties of models, such as correctness, minimality, traceability, consistency, and unambiguity are subsumed by their framework [9]. In our application of the framework, we had to extend the definition of pragmatic quality to cover model creation and modification in addition to the already addressed comprehension aspect. Furthermore, we apply this framework at three levels: FSMs, FSMLs, and FSMF.

SEQUAL [10] is a comprehensive semiotic quality framework for models that grew out of the framework by Lindland et al. Additionally to syntactic, semantic, and pragmatic qualities, SEQUAL proposes four more kinds of qualities: empirical, perceived semantic, social, and organizational. Empirical quality is related to properties that can be empirically perceived by just looking at a model and without considering its syntax or semantics (e.g., use of colors, quality of layout, presence of redundancy). Perceived semantic quality relates the model semantics to what the audience already knows about the model domain and how that may influence their model understanding. Social quality is related to how differently a given model may be interpreted by different members of the audience. Furthermore, the quality is related to the ability to express such differences and to support consensus building. Means to improve social quality include model comparison and model merging facilities. In this paper, we only considered the organizational quality. We chose not to consider empirical quality since we view it as highly overlapping with the pragmatic quality. Furthermore, we feel that the concept of perceived semantic quality needs further elaboration and concretization to be readily applicable in our context; however, we recognize the relevance of social quality to FSMLs, since FSMLs may have multiple simultaneous users of different backgrounds, e.g., application developers, architects, and quality assurance

engineers. We leave this aspect for future work.

The Cognitive Dimensions (CDs) framework has been previously used to assess the pragmatic quality of modeling languages (e.g., [66]) and DSLs (e.g., [67]). Blackwell at al. [68] provided criteria for extending CDs with new dimensions. User studies with FSMLs may well reveal that the CDs framework may need further adaptation and extension to the context of FSMLs.

8.2 Modularization of crosscutting concerns

Aspect-Oriented Programming (AOP). The goal of AOP is to modularize the implementation of crosscutting concerns [69]. Similar to AOP, FSMLs also address the issue of understanding and maintaining crosscutting concerns. In particular, FSMLs help to locate the implementation of framework-provided concepts and features that may be scattered across the application code and tangled with the implementation of other concepts. Interestingly, the FSML mapping types used for defining behavioural patterns are essentially pointcut language constructs. Some of the mapping types correspond to AspectJ constructs and other, such as data-flow related mapping types, go beyond AspectJ [14].

In contrast to AOP languages such as AspectJ, FSMLs do not separate the scattered code fragments into separate aspect modules, but leave them inline and support their localization through traceability links. This approach is less invasive for existing applications that do not use an aspect language. Also, it can be argued that not all crosscutting code is appropriate to be separated out [70]. A hybrid approach that either leaves the crosscutting separated or inline and allows to freely switch between the two is also possible (e.g., [71]). Exploring such hybrid strategies in the context of FSMLs is future work.

Feature-Oriented Programming (FOP). The goal of FOP approaches such as AHEAD [72] is to allow applications to be synthesized from different configurations of features. Features are increments in program functionality and FOP seeks to provide mechanisms to encapsulate features as first-class modules. FOP and FSMLs are similar in that they both use feature models to express the configurability of features and they both consider the mapping of features to code and other artifacts such as XML configuration files; however, there are two important differences. First, FOP specifically targets product-line engineering and seeks to synthesize entire applications in an application domain. That is, each application is fully described by a particular feature configuration. In contrast, FSMs model only one aspect of framework-based applications, namely how the framework-provided concepts and features are instantiated in the application code. In particular, applications are likely to contain significant amounts of code that is application-specific and is not reflected in their FSMs. Second, FOP focuses on the modularity and compositionality of features and it exploits their algebraic properties in application synthesis and optimization. That is, FOP helps to create highly modular application and product-line architectures. As a result, existing applications need to be re-factored into appropriate features in order to take advantage of FOP [73]. In contrast, FSMLs focus on the ability to automatically reverse engineer models from applications that use the framework for which the FSML was designed. Furthermore, both the application code and the FSMs can be freely edited and automatically kept in sync through round-trip engineering. Thus, FSMLs do not impose any architectural constraints on applications beyond what is required by the corresponding framework. Nevertheless, both approaches are complementary and exploring synergies between them is future work.

8.3 Supporting framework-based application development

Documentation-oriented approaches. Early approaches to supporting framework instantiation focused on providing API documentation in the form of cookbooks [74], which outlined development tasks and steps supplemented with code examples. Pree et al. showed how cookbooks could be made active and how development steps could automated [75]. Ortigosa et al. went further by using intelligent agent technology to interactively guide the user in making implementation choices based implementation recipes [76].

In 1992, Johnson proposed documenting frameworks using patterns [77]. The patterns described the purpose and the design of a framework and common ways of using it.

Fontoura et al. proposed a special profile of the UML language for representing framework APIs called UML-F [78]. The profile defines annotations that can be placed on API elements to indicate their function. The users specialize a UML-F model to produce the design of their application and a supporting tool transforms the model into code. The tool has the form of a wizard that can only generate code, i.e., it cannot be used for maintaining the consistency between the model and the code.

Pattern-based approaches. Hakala et al. proposed a pattern-based approach to generating task-driven programming environments for frameworks [79]. In this approach, specialization patterns are used to describe extension points of frameworks. A tool called FRamework EDitor (FRED) is used by application developers in the process of *casting*, i.e., instantiating the given specialization pattern in the code. FRED supports iterative and incremental casting which reflects the natural way programmers work.

Tourwe proposed documenting framework's design using design patterns and then using such documentation to provide active support in framework-based software evolution [80]. After manually identifying design pattern instances used in the framework, the associated tool can support framework evolution and instantiation related to these pattern instances. The tool can create

subclasses of framework API classes and add empty method declarations in the application code to support framework instantiation. Application developers are expected to provide implementation for the created method bodies.

More recently, Fairbanks proposed encoding common framework API usage patterns as *design fragments* [23]. Such a design fragment has a name, a goal that can be accomplished by implementing it, description of classes, methods, fields, and method and constructor calls that need to be created to accomplish the goal, and description of the relevant parts of the framework's API. Developers choose a design fragment they wish to implement and manually bind it to their code allowing the tool to check the correctness of the implementation.

The FSML approach differs from pattern-based approaches by emphasizing a more conceptual view of the API as a language rather than focusing on individual API usage patterns. In particular, FSMLs can include higher-level domain concepts, which are then decomposed into individual implementation patterns. Similarly to the pattern approaches, in which the patterns have to be identified and specified manually, FSMLs have to be built manually; however, FSMLs support automatic feature instance location in the application code and the addition of feature instance implementations to code at the level of individual statements (rather than class skeletons only). Furthermore, through variability mechanisms, FSMLs can encode and organize many alternative ways of implementing a single concept, each of which would require a separate design pattern or fragment.

Framework API usage comprehension and pattern mining. The pattern-based approaches discussed previously assume that the patterns are identified and specified manually. Manual identification of such patterns in sample applications can be a difficult task because of the potentially large amount of application code that needs to be processed. Several tool-supported approaches have been proposed to ease this task.

Specification mining is concerned with extracting API call protocols form application code. These approaches use either dynamic analysis, e.g., [81], [82], [83], [84], or static analysis, e.g., [85], [86], [87], to extract API call sequences from sample applications, and then apply mining techniques to generalize the sequences to generic protocol specifications. Interestingly, these approaches mainly focus on library APIs rather than framework APIs. The relevance of API protocols to libraries is obvious: library components, such as network connections and cashes, require their users to observe lifecycle protocols, such as those involving component creation, initialization, actual processing, tear-down, and destruction. In contrast, frameworks tend to enforce protocols themselves using callbacks, and the framework API users only need to know which framework services should be called in which callback. Therefore, protocol mining seems less useful for frameworks than libraries.

Several approaches deal with mining for specific framework API usage patterns. For example, Prospector [88] and PARSEWeb [89] mine for API call sequences needed to arrive at some API type (class or interface) given another API type. XSnippet [90] mines for code fragments instantiating a given API type. All these approaches require the API user to issue a query, upon which one or more code fragments based on the sample applications is produced.

Strathcona [91] and FrUiT [92] also search existing sample code for API usages, but they obviate the need to state queries explicitly. Instead, they use the current code under development as a query. Strathcona locates relevant code in sample applications based on heuristically matching the structure of the code under development to the example code, where structure covers information such as type hierarchies and method call graphs. FrUiT mines sample code for frequent API usage patterns as association rules, e.g., *subclass* $A \Rightarrow$ *call* m, which is similar to the earlier CodeWeb approach [93]. In contrast to CodeWeb, however, FrUiT uses such rules to suggest implementation steps for a Java class under development.

FUDA [94], [95] allows developers to extract entire implementation templates for concepts of interest. Instead of specifying a code-oriented query, the developer uses a tracer to collect two execution traces of the concept of interest, e.g., a context menu, from one or two applications. The traces can be optionally marked with the beginning and the end of concept execution. Based on these traces, FUDA Analyzer generates a Java-like concept implementation template by intersecting the traces and statically extracting further information from the sample application code. A limitation of FUDA is its inability to detect implementation variants: the implementation template is useful for implementing a single variant of a concept.

FSML engineering can clearly benefit from tool support to mine existing application code for API usage patterns: such mining would allow the discovery of new concepts and features for the FSML under construction. During the development of the exemplar FSMLs, we used an existing FSML version to reverse engineer sample code. We then followed the traceability links from the extracted FSMs to code and also used Eclipse's search capabilities to manually discover missed patterns. However, a proper pattern mining tool could make this process more systematic by completely scanning the framework-application boundary and by organizing the discovered patterns into feature models annotated with pattern occurrence statistics. Mining probabilistic feature models from sets of feature configurations obtained by static program analysis [96] is a step towards such tool support. Further exploration of API usage mining for FSML engineering is future work.

Framework API constraint checking. Hou and Hoover proposed an approach to checking applications for adherence to structural API constraints specified in *Structural Constraint Language* (SCL) [97], [98]. The ap-

proach was demonstrated for C++ and Java frameworks. In addition to basic 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. FSMLs go beyond API constraint checking by representing and organizing these constraints as a language and supporting forward and reverse engineering.

Framework API migration. Two categories of framework API migration can be distinguished: upgrading application code from an older API version to a newer one and migrating application code from one framework to a different one.

There has been recently a flurry of work in the first category, e.g., recording API refactorings and replying them on the application code to be updated [99], inferring such refactorings by comparing different API versions [100], or inferring them from already migrated applications [101]. We have not encoded upgrades using FSMLs; however, upgrades should be feasible using a single metamodel since they require relatively simple code changes.

In the second category, we already mentioned migration from Struts to Java Server Faces using FSMLs [25] (cf. Box 17). Another example of API migration between different frameworks is *ajaxification*, i.e., migration of multi-page web applications to Asynchronous JavaScript and XML (AJAX) [102]. In AJAX, 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, the authors proposed 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. This migration process could be viewed as applying RE, MA, and FE. Exploring the above and additional migration scenarios using FSMLs is future work.

9 Conclusion

The main contributions of this paper are four exemplar FSMLs and the experience of building these languages presented in the form of an engineering method. The method provides FSML developers with a detailed and comprehensive set of steps, guidelines, and examples.

The development of the FSML approach is grounded in the design science paradigm and the method represents a necessary step in the maturation of the FSML approach. The method was constructed by specializing well-established approaches: feature-oriented domain analysis (FODA), the model quality framework of Lindland et al., the Cognitive Dimensions (CDs) framework, and some elements of the Unified Process. Specializing these approaches to the FSML context allowed us to systematically elicit, structure, and document our experience. We have justified the method by tracing its steps back to the development of the exemplar FSMLs and to the elements of the base approaches.

In the course of the method development, FODA had to be extended to handle cardinality-based feature models and the special FSML modeling constructs such as essentiality, keys, and mapping definitions. The model quality framework was applied at three levels: framework-specific models, modeling languages, and the modeling foundation, while distinguishing between the pragmatic quality of a language extension and its metamodel (cf. Figure 5). The CDs framework was used to evaluate the pragmatic quality of the exemplar FSMLs (cf. Box 6). Furthermore, the development of the exemplar FSMLs showed that a single FSML metamodel can be used to support multiple use cases and that metamodel changes needed to support each additional use cases were mostly additive.

Future work includes performing user studies with the exemplar FSMLs and the method, extending round-trip capabilities with code removal and refactoring capabilities, and formalizing the overall approach.

We believe that the paper provides a strong case for the benefits of modeling framework APIs as modeling languages by showing how such languages can support API understanding and completion code understanding, analysis, creation, evolution, and migration. The FSML approach can be categorized as model-supported engineering, in which models are supporting artifacts. This is in contrast to model-driven engineering, where models are the primary source artifacts. FSMs can always be extracted using RE and discarded if no longer needed. Furthermore, FSMs can be always kept up-to-date through RTE, regardless whether the code or the language definition has changed. In the latter case, after adding new features to the FSML, existing FSMs can simply be updated with instances of these new features matched in the code through RTE. Therefore, incremental development of FSMLs as prescribed by the method also supports incremental adoption of such languages in practice with minimal changes to the existing development processes.

Acknowledgments

The authors would like to thank Herman Lee for implementing the XML mapping interpreter used by the Struts and EJB FSMLs and Henry Lau for implementing some of the code transformations for mapping types related to Java. We also thank the anonymous reviewers for their valuable comments. This research was supported by IBM Centers for Advanced Studies in Ottawa and Toronto, the Natural Sciences and Engineering Research Council of Canada, and the Ontario Research Fund.

References

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

[2] D. Springgay, *Creating an Eclipse View*, November 2001, http://www.eclipse.org/articles/viewArticle/ViewArticle2.html.

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

[4] M. Antkiewicz and K. Czarnecki, "Framework-specific modeling languages with round-trip engineering," in *MoDELS*, ser. LNCS, vol. 4199, 2006, pp. 692–706.

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

[6] I. Jacobson, G. Booch, and J. Rumbaugh, *The unified software development process*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[7] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," Software Engineering Institute, Carnegie Mellon University, Tech. Rep. CMU/SEI-90TR -21, 1990.

[8] A. R. Hevner, S. T. March, J. Park, and S. Ram, "Design science in information systems research," *MIS Quarterly*, vol. 28, no. 1, pp. 75–105, 2004.

[9] O. I. Lindland, G. Sindre, and A. Sølvberg, "Understanding quality in conceptual modeling," *IEEE Softw.*, vol. 11, no. 2, pp. 42–49, 1994.

[10] J. Krogstie, G. Sindre, and H. Jørgensen, "Process models representing knowledge for action: a revised quality framework," *Eur. J. Inf. Syst.*, vol. 15, no. 1, pp. 91–102, 2006.

[11] T. R. G. Green, "Cognitive dimensions of notations," in *People and Computers V*, A. Sutcliffe and L. Macaulay, Eds., 1989, pp. 443–460.

[12] T. R. G. Green and M. Petre, "Usability analysis of visual programming environments: A 'cognitive dimensions' framework," *J. Vis. Lang. Comput.*, vol. 7, no. 2, pp. 131–174, 1996.

[13] S. T. March and G. F. Smith, "Design and natural science research on information technology," *Decis. Support Syst.*, vol. 15, no. 4, pp. 251–266, 1995.

[14] M. Antkiewicz, T. Tonelli Bartolomei, and K. Czarnecki, "Fast extraction of high-quality framework-specific models from application code," *J. Autom. Soft. Eng.*, vol. 16, no. 1, pp. 101–144, Mar. 2009.

[15] K. Czarnecki, S. Helsen, and U. Eisenecker, "Formalizing cardinality-based feature models and their specialization," *Software Process Improvement and Practice*, vol. 10, no. 1, pp. 7–29, 2005, special issue on Software Variability: Process and Management.

[16] M. Stephan and M. Antkiewicz, "Ecore.fmp: A tool for editing and instantiating class models as feature models," ECE, Univeristy of Waterloo, Tech. Rep. #2008-08, 2008, http://gp.uwaterloo.ca/tr/2008-stephan-ecore-fmp.pdf.

[17] K. Czarnecki and U. W. Eisenecker, *Generative programming: methods, tools, and applications*. Addison-Wesley Publishing Co., 2000.

[18] M. Antkiewicz and K. Czarnecki, "Design space of heterogeneous synchronization," in *GTTSE*, ser. LNCS, vol. 5235, 2008, pp. 3–46.

[19] S. Khanna, K. Kunal, and B. C. Pierce, "A formal investigation of diff3," in *FSTTCS*, 2007.

[20] J. Makhoul, F. Kubala, R. Schwartz, and R. Weischedel, "Performance measures for information extraction," in *Proceedings of DARPA Broadcast News Workshop*, 1999, pp. 249–252.

[21] W. Nöth, *Handbook of Semiotics*. Bloomington: Indiana University Press, 1996.

[22] M. Stephan, "Discovery of Java EE 5 EJB antipattern instances using framework-specific models," Master's thesis, University of Waterloo, 2009, in preparation.

[23] G. Fairbanks, D. Garlan, and W. Scherlis, "Design fragments make using frameworks easier," in *OOPSLA*, 2006, pp. 75–88.

[24] B. Dudney, S. Asbury, J. Krozak, and K. Wittkopf, *J2EE AntiPatterns*. Wiley, 2003.

[25] A. P. Cheema, "Struts2JSF - framework migration in J2EE using Framework-Specific Modeling Languages," Master's thesis, University of Waterloo, 2007, http://hdl.handle.net/10012/3031.

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

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

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

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

[30] "Struts 1.35 dtd," Apache Software Foundation, July 2006, http://struts.apache.org/1.3.5/dtds/struts-config_1_3.dtd.

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

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

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

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

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

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

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

[38] E. Jendrock, J. Ball, D. Carson, I. Evans, S. Fordin, and K. Haase, *The Java EE 5 Tutorial*, Sun Microsystems, Inc., September 2007, http://java.sun.com/javaee/5/docs/tutorial/doc/index.html.

[39] K. Lee and K. C. Kang, "Feature dependency analysis for product line component design," in *ICSR*, ser. LNCS, no. 3107, 2004, pp. 69–85.

[40] M. Antkiewicz, T. Tonelli Bartolomei, and K. Czarnecki, "Automatic extraction of framework-specific models from framework-based application code," in *ASE*, 2007, pp. 214–223.

[41] A. Blackwell and T. Green, "A cognitive dimensions questionnaire," February 2007, http://www.cl.cam.ac.uk/~afb21/CognitiveDimensions/CDquestionnaire.pdf, last accessed Feb. 20, 2009.

[42] D. L. Parnas and P. C. Clements, "A rational design process: How and why to fake it," *IEEE Trans. Softw. Eng.*, vol. 12, no. 2, pp. 251–257, 1986.

[43] D. Hou, K. Wong, and H. J. Hoover, "What can programmer questions tell us about frameworks?" in *IWPC*, 2005, pp. 87–96.

[44] F. Shull, F. Lanubile, and V. R. Basili, "Investigating reading techniques for object-oriented framework learning," *IEEE Trans. Softw. Eng.*, vol. 26, no. 11, pp. 1101–1118, 2000.

[45] M. Antkiewicz and K. Czarnecki, "Framework-specific modeling languages; examples and algorithms," ECE, U. of Waterloo, Tech. Rep. 2007-18, 2007.

[46] H. Lee, M. Antkiewicz, and K. Czarnecki, "Towards a generic infrastructure for framework-specific integrated development environment extensions," in *DSPD*, 2008.

[47] G. Little and R. C. Miller, "Keyword programming in Java," *Automated Software Engineering*, vol. 16, no. 1, pp. 37–71, march 2008.

[48] D. Harel and B. Rumpe, "Meaningful modeling: What's the semantics of "semantics"?" *Computer*, vol. 37, no. 10, pp. 64–72, 2004.

[49] P. Avgustinov, E. Hajiyev, N. Ongkingco, O. de Moor, D. Sereni, J. Tibble, and M. Verbaere, "Semantics of static pointcuts in aspectj," *SIGPLAN Not.*, vol. 42, pp. 11–23, 2007.

[50] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble, "Adding trace matching with free variables to AspectJ," in *OOPSLA*, 2005, pp. 345–364.

[51] M. Antkiewicz and K. Czarnecki, "FeaturePlugin: feature modeling plug-in for eclipse," in *Eclipse Technology eXchange Workshop*, 2004, pp. 67–72.

[52] K. Czarnecki and P. Kim, "Cardinality-based feature modeling and constraints: A progress report," in *Proceedings of the International Workshop on Software Factories, OOPSLA*, 2005.

[53] K. Czarnecki and A. Wasowski, "Feature diagrams and logics: there and back again," in *SPLC*, 2007, pp. 23–34.
[54] D. Roberts and R. Johnson, "Evolving frameworks: A pattern language for developing object-oriented frameworks," in *PLoP*, 1996.
[55] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM Comput. Surv.*, vol. 37, no. 4, pp. 316–344, 2005.
[56] M. Bravenboer and E. Visser, "Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions," in *OOPSLA*, 2004, pp. 365–383.
[57] T. Stahl and M. Voelter, *Model-Driven Software Development*, 1st ed. Wiley, May 2006.
[58] T. Clark, P. Sammut, and J. Willans, *Applied Metamodeling. A Foundation for language driven development*, 2nd ed. Ceteva, 2008.
[59] J. Greenfield, K. Short, S. Cook, and S. Kent, *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, 2004.
[60] "Meta object facility (mof) core specification v. 2.0," Object Management Group, Inc., January 2008, http://www.omg.org/mof/.
[61] F. Budinsky, S. A. Brodsky, and E. Merks, *Eclipse Modeling Framework*. Pearson Education, 2003.
[62] S. Cook, G. Jones, S. Kent, and A. C. Wills, *Domain-Specific Development with Visual Studio DSL Tools*. Addison-Wesley Professional, 2007.
[63] K. Smolander, "GOPRR: a proposal for a meta level model," 1993, University of Jyväskylä, Finland.
[64] S. Kelly and J.-P. Tolvanen, *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE Computer Society Press, 2008.
[65] D. L. Moody, "Theoretical and practical issues in evaluating the quality of conceptual models: current state and future directions," *Data Knowl. Eng.*, vol. 55, no. 3, pp. 243–276, 2005.
[66] M. Kutar, C. Britton, and T. Barker, "A comparison of empirical study and cognitive dimensions analysis in the evaluation of UML diagrams," in *Workshop of the Psychology of Programming Interest Group*, 2002, pp. 1–14.
[67] C. A. Austin, "Renaissance: a functional shading language," Master's thesis, Iowa State University, 2005.
[68] A. F. Blackwell, C. Britton, A. L. Cox, T. R. G. Green, C. A. Gurr, G. F. Kadoda, M. Kutar, M. Loomes, C. L. Nehaniv, M. Petre, C. Roast, C. Roe, A. Wong, and R. M. Young, "Cognitive dimensions of notations: Design tools for cognitive technology," in *CT*, 2001, pp. 325–341.
[69] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *ECOOP*, 1997, pp. 220–242.
[70] S. Apel and D. Batory, "When to use features and aspects?: a case study," in *GPCE*, 2006, pp. 59–68.
[71] C. H. P. Kim, K. Czarnecki, and D. Batory, "On-demand materialization of aspects for application development," in *SPLAT*, 2008, pp. 1–6.
[72] D. Batory, J. N. Sarvela, and A. Rauschmayer, "Scaling step-wise refinement," in *ICSE*, 2003, pp. 187–197.
[73] J. Liu, D. Batory, and C. Lengauer, "Feature oriented refactoring of legacy applications," in *ICSE*, 2006, pp. 112–121.
[74] G. E. Krasner and S. T. Pope, "A cookbook for using the model-view controller user interface paradigm in Smalltalk-80," *J. Object Oriented Program.*, vol. 1, no. 3, pp. 26–49, 1988.
[75] W. Pree, G. Pomberger, A. Schappert, and P. Sommerlad, "Active guidance of framework development," *Software - Concepts and Tools*, vol. 3, no. 16, 1995.
[76] A. Ortigosa and M. Campo, "Smartbooks: A step beyond active-cookbooks to aid in framework instantiation," in *TOOLS*, 1999, p. 131.
[77] R. E. Johnson, "Documenting frameworks using patterns," in *OOPSLA*, 1992, pp. 63–76.
[78] M. Fontoura, W. Pree, and B. Rumpe, "UML-F: A modeling language for object-oriented frameworks," in *ECOOP*, 2000, pp. 63–82.
[79] M. Hakala, J. Hautamäki, K. Koskimies, J. Paakki, A. Viljamaa, and J. Viljamaa, "Generating application development environments for java frameworks," ser. LNCS, vol. 2186, 2001, pp. 163–176.
[80] T. Tourwé, "Automated support for framework-based software evolution," Ph.D. dissertation, Vrije Universiteit Brussel, 2002.
[81] G. Ammons, R. Bodík, and J. R. Larus, "Mining specifications," in *POPL*, 2002, pp. 4–16.
[82] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das, "Perracotta: Mining temporal API rules from imperfect traces," in *ICSE*, 2006, pp. 282–291.
[83] D. Lo, S.-C. Khoo, and C. Liu, "Efficient mining of iterative patterns for software specification discovery," in *KDD*, 2007, pp. 460–469.
[84] S. Sankaranarayanan, F. Ivanči, and A. Gupta, "Mining library specifications using inductive logic programming," in *ICSE*, 2008, pp. 131–140.
[85] A. Wasylkowski, A. Zeller, and C. Lindig, "Detecting object usage anomalies," in *FSE*, 2007.
[86] M. Acharya, T. Xie, J. Pei, and J. Xu, "Mining API patterns as partial orders from source code: From usage scenarios to specifications," in *FSE*, 2007.
[87] M. K. Ramanathan, A. Grama, and S. Jagannathan, "Path-sensitive inference of function precedence protocols," in *ICSE*, 2007, pp. 240–250.
[88] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, "Jungloid mining: Helping to navigate the API jungle," in *PLDI*, 2005, pp. 48–61.
[89] S. Thummalapenta and T. Xie, "PARSEWeb: A programmer assistant for reusing open source code on the web," in *ASE*, 2007, pp. 204–213.
[90] N. Sahavechaphan and K. Claypool, "XSnippet: Mining for sample code," in *OOPSLA*, 2006, pp. 413–430.
[91] R. Holmes and G. C. Murphy, "Using structural context to recommend source code examples," in *ICSE*, 2005, pp. 117–125.
[92] M. Bruch, T. Schafer, and M. Mezini, "FrUiT: IDE support for framework understanding," in *ETX*, 2006, pp. 55–59.
[93] A. Michail, "Data mining library reuse patterns using generalized association rules," in *ICSE*, 2000, pp. 167–176.
[94] A. Heydarnoori, "Supporting framework use via automatically extracted concept-implementation templates," Ph.D. dissertation, University of Waterloo, Canada, 2009.
[95] A. Heydarnoori, K. Czarnecki, and T. Tonelli Bartolomei, "Supporting framework use via automatically extracted concept-implementation templates," in *ECOOP*, 2009, submitted for review.
[96] K. Czarnecki, S. She, and A. Wasowski, "Sample spaces and feature models: There and back again," in *SPLC*, 2008, pp. 22–31.
[97] D. Hou and H. Hoover, "Using SCL to specify and check design intent in source code," *IEEE Tran. Soft. Eng.*, vol. 32, no. 6, pp. 404–423, June 2006.
[98] D. Hou, "FCL: Automatically detecting structural errors in framework-based development," Ph.D. dissertation, University of Alberta, 2004.
[99] D. Dig and R. Johnson, "The role of refactorings in API evolution," in *ICSM*, 2005, pp. 389–398.
[100] Z. Xing and E. Stroulia, "API-evolution support with Diff-CatchUp," *IEEE Tran. Soft. Eng.*, vol. 33, pp. 818–836, 2007.
[101] T. Schäfer, J. Jonas, and M. Mezini, "Mining framework usage changes from instantiation code," in *ICSE*, 2008, pp. 471–480.
[102] A. Mesbah and A. van Deursen, "Migrating multi-page web applications to single-page AJAX interfaces," *CSMR*, pp. 181–190, 2007.

Michał Antkiewicz is a Postdoctoral Fellow at the University of Waterloo.

Krzysztof Czarnecki is an Associate Professor at the University of Waterloo.

Matthew Stephan is a MASc. Candidate at the University of Waterloo.