Uniform Support for Modeling Crosscutting Structure

by

Maria A. Tkatchenko

B.Sc., University of British Columbia, 2004

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

The University of British Columbia

April 2006

Abstract

We propose bottom-up support for modeling crosscutting structure in UML by adding a simple join point model to the meta-model. This supports built-in crosscutting modeling constructs such as class and sequence diagrams, collaborations, and state machines. It also facilitates adding new kinds of crosscutting modeling constructs such inter-type declarations and advice.

A simple planner tool produces a uniform representation of the crosscutting structure, which can then be displayed or analyzed in a variety of ways. We demonstrate a couple of simple automated analysis tools which take advantage of the exposed crosscutting structure. We also discuss how support for advice could be added to the meta-model and planner, and the semantic differences between advice in UML and AspectJ.

Contents

List of Tables

List of Figures

Acknowledgements

There are many people without whose support this would not have been possible, or at least would have taken much longer than it has.

First and foremost, I'm grateful to my supervisor, Gregor Kiczales, for all the time and effort he has put into working with me on this, both when brainstorming ideas and writing the submission for the conference. Thanks for giving me the opportunity to work on this project, and present it at conferences — this was definitely an educational and rewarding experience. I hope you keep finding time to play with your students' code despite all of your other responsibilities!

Thanks to Eric Wohlstadter, who generously agreed to be my second reader when I came to him in a panic after realizing that I needed one.

I should also thank both the employees and fellow CAS students at the IBM-Ottawa Palladium lab, who are too numerous to name here, and who shared with me their feedback and ideas, which helped shape this project. You made my time in Ottawa one of my best summers away from Vancouver.

Finally, thanks to my parents, for always being there for me and encouraging me to go on and look for bigger and better things, both in life and education. Thanks to all of my friends, and Dan, for your endless patience and your tireless attempts at distraction when I got too bogged down in work. I'm glad you're all a constant presence in my life. Thanks for all the fun times we had, and new sports you have introduced me to. My time spent with all of you has definitely been a rewarding experience in its own right!

MARIA A. TKATCHENKO

The University of British Columbia
April 2006

Chapter 1

Introduction

The Unified Modeling Language (UML) provides support for modeling a system from different perspectives [24]. Some of these perspectives have a hierarchical relationship to each other, such as package and class diagrams. Others have a crosscutting relationship [19, 23], whereby a given element may appear in both diagrams, with each diagram only partially specifying the element. For example, a sequence diagram can crosscut a class diagram, in that it may include calls to methods from multiple classes [14, 10]; collaboration diagrams can crosscut class and sequence diagrams [33]; statecharts can crosscut all the others. There have also been proposals to extend UML with new crosscutting modeling constructs such as aspects, advice, inter-type declarations (ITDs) and role bindings [1, 16, 20, 26, 32, 35, 39].

Modeling in UML has gotten a boost from the adoption by the OMG of standards for the specification of notation and semantics, and the development of open-source modeling frameworks such as EMF [29] and UML2 [30]. Such frameworks enable the creation of more task-specific, less bloated modeling tools than what is generally available commercially. Despite the growing popularity of UML, it still has a number of strong drawbacks, in particular the misalignment between the feature-oriented requirements and object-oriented design and code, as discussed by Clarke et. al. in [5].

Aspect-oriented programming (AOP) is an area that focused initially on support for crosscutting in code. In code, crosscutting can cause a single concept to have to be implemented in multiple classes and multiple locations in code, because of the way the system has been modularized. AOP enables such concerns to be implemented in a modular rather than scattered and tangled fashion. With modeling, we take scattering to mean that the model-level implementation of a single concept is spread across multiple diagrams of the same kind (e.g. many methods in a class diagram). But because UML already provides crosscutting diagram kinds, crosscutting structure can already be modularized in UML (e.g. a sequence diagram and a class diagram can modularize concerns that crosscut each other).

Our work was partially motivated by observing the difficulty of adding pattern composition support to existing UML tools. Prior work in AOP has shown that many patterns are easier to implement using AspectJ [9]. We show how we can achieve the same kind of benefits from aspect-orientation for patterns during modeling by introducing a *join point model* (JPM) to the UML meta-model. We also present a way in which we can support other forms of crosscutting structure in UML. So rather than using modeling to support AOP [15, 16], our focus is on using the central mechanism of AOP to support modeling.

In this work we show that the modeling of crosscutting relationships within UML diagrams such as those mentioned above can be supported by using a join point model. We propose bottom-up support for crosscutting in UML, by adding a simple JPM to the UML meta-model. Our enhanced meta-model supports display and automated analysis of both pre-existing and new forms of crosscutting structure between elements in UML diagrams. We implement a simple tool (which we call the planner) which exposes the crosscutting structure in the model. Through a number of examples, we show how the JPM makes crosscutting structure explicit and simplifies implementation of analysis tools.

In AOP, weaving is defined as the coordination of interactions between the

crosscutting concerns. In AOP languages like AspectJ, this involves ensuring that advice runs when it should and inserting inter-type declarations at their target locations. This is done in two phases, the planner followed by the munger — the planner identifies the actual join points at which the concerns crosscut, and the munger implements the interaction (e.g. by modifying execution, modifying code, or modifying the model).

In this thesis, we present a planner tool that provides simple coordination of crosscutting structure in UML models. By providing a uniform representation of the interactions between crosscutting elements, our planner makes it easier to implement model analysis and display tools. The planner records its results by associating with each model element a set of all the other model elements with which it crosscuts. Once the pair-wise crosscutting structure is collected into these sets, it can be analyzed and presented in a variety of ways.

In order to understand crosscutting relationships as they occur in modeling, and how they affect the design of the system as a whole, we need to be able to represent the relationships between diagrams in a more coherent way. UML models — in particular large, industrial-sized UML models — are often difficult to work with, at least in part due to the lack of information in one diagram on the behaviour or structure defined in another. What we would like to see are tools that help the modeler see the big picture, and the place of specific elements within it. This work presents a framework and a planner tool that would simplify implementation of analysis tools that would help the modeler see the big picture. We also discuss two simple analysis tools that we have implemented. We do not propose a graphical notation for showing crosscutting relationships; the crosscutting relationships exposed by our planner can be displayed in any number of ways, and examining which is better is left to future work.

The contributions of this work are to show that (i) the crosscutting structure of several traditional modeling relationships, as well as newer aspect-oriented mod-

eling relationships, can be supported by a meta-model enhanced with a simple JPM, (ii) traversing the model to collect the crosscutting structure (planning) is straightforward, and (iii) the combination of (i) and (ii) makes it easy for modeling tool implementers to create tools that access, analyze and display crosscutting relationships of interest. We also present (iv) a planner tool which exposes the crosscutting structure in the model for use by the tools mentioned in (iii).

The rest of this document is structured as follows. Chapter 2 goes into detail about the related work, setting up the context for our work. Chapter 3 provides a background on AOP and explains the major concepts that will be used in this thesis. Chapter 4 introduces a couple of examples, used in the subsequent chapters. Chapter 5 describes the additions we made to the meta-model, as well as the reasons behind these choices. Chapter 6 talks about the implementation of our planner tool. Chapter 7 uses the examples introduced in Chapter 4 to show how our planner would help answer some questions we thought would be pertinent to model analysis. Chapter 8 concludes the discussion with a summary and some suggestions for future work.

Chapter 2

Related Work

There are two streams of research related specifically to our work. The first looks at explicitly adding aspects or AOP concepts to UML, either by using the extension mechanisms provided in UML, or by changing the UML meta-model directly. The second looks at improving the design process by allowing the modeler to decompose the design into independent components, and then providing means to compose the different components or diagrams, which helps alleviate problems associated with some forms of crosscutting.

2.1 Adding Support for AOP to UML

2.1.1 Standard Extension Mechanisms

The following proposals use the standard extension mechanisms provided by UML to support aspects or aspect-oriented extensions.

Stein et al. [35] introduce the concept of weaving to the extended meta-model of UML through the use of stereotypes. The work deals with both structural crosscutting in the form of introductions, and behavioural crosscutting in the form of advice. The behavioural crosscutting case is closest to our work, so we will focus on it in our discussion. The base (crosscut) and advice (crosscutting) behaviours,

specified by sequence diagrams, can be merged to display the final expected behavior. This involves splitting apart the base sequences at the join points, and later composing the sequence with the crosscutting behaviour included at each of the join points affected. The set of calls in a sequence may need to be totally ordered for weaving to guarantee preservation of behaviour, which is computationally expensive. The authors use stereotypes to imitate the advice and pointcut constructs of AspectJ, and explicit weaving instructions to specify composition of behaviours, thus forcing the modeler to think at a lower level of abstraction.

Pawlak presents a UML notation for designing aspect-oriented applications [26]. In this work, he introduces three new concepts to UML: groups, pointcut relations, and aspect-classes. Groups are used to specify base objects in an aspect-oriented context. Pointcut relations define crosscuts within the program, linking aspect-methods to points in the base class. Aspect-classes implement extensions of the base program semantics at points denoted by pointcut relations, and contain both regular and aspect-methods. Stereotypes are used to support the use of all of these new modeling elements.

Ho and Jezequel present a toolkit for building application-specific weavers for generating detailed design models from high-level aspect-oriented UML models [12]. These weavers are implemented as model transformations — each weaving step is a transformation applied to a UML model. Transformations are specified by the designer by explicitly composing a set of operators available from the toolkit. The authors use built-in extension mechanisms in UML, namely stereotypes, tag values, and design pattern occurrences, to add non-functional information or crosscutting behavior to base model elements. This information is then read by the appropriate weaver, and applied during the relevant weaving step.

Jezequel and Plouzeau discuss how features of a UML model can be organized around the notions of quality of service contracts and aspects [15]. Contracts are modeled in UML using a small set of stereotypes, and specify non-functional

properties. Aspects are represented using parameterized collaborations and transformation rules, and specify how contracts can be implemented. Unlike the above work, the authors here take the approach that in order for aspects to be reusable, the join points have to be specified separately. Now there are three entities: the target model, aspect, and join point definition. OCL is used to specify the transformations that take place during weaving.

Suzuki and Yamamoto add new elements for the aspect and woven class using stereotypes, and reuse an existing element for the aspect-class relationship [38]. An aspect is shown as a class with an "aspect" stereotype, and may contain operations with the stereotype "weave", which can represent either introductions or advice. They use the realization relationship to represent the aspect-class dependency. Classes with aspects woven into them are shown using the "woven class" stereotype. The main contribution of this work, however, is the development of the UXF/a, an extension to the UXF (UML eXchange Format), an XML-based language for describing UML models. With UXF/a a modeler can add aspect information to models, and the authors have developed translators that allow UML aspect models to be shown in popular CASE tools such as Rational Rose and MagicDraw.

2.1.2 Meta-Model Changes

The work discussed in this section uses extensions to the UML meta-model to support aspects or aspect-oriented extensions.

Kande argues that aspects need to be first-class elements in UML [16]. His work is important to us because it shows that others have considered, and successfully argued for, the approach of introducing weaving or crosscutting as a basic concept in UML, instead of using extension mechanisms. He claims that the composition of a standard UML model with an aspect model does not do a good job of modularizing the separate concerns — the elements in the design model are coupled more than they would be in the code. In addition, the composed model does not

communicate the ability to plug and un-plug aspects from the core functionality. He shows that the main reason for this is that since UML doesn't include the concept of weaving, the concerns that are well-separated in the AO program end up being scattered throughout the design model. Thus, a new model element which encapsulates the specification of the aspect as well as models the interaction between all crosscutting objects may be needed.

Citing the restrictions that arise when using stereotypes and profiles to extend UML, Lions proposes introducing AOP into UML at the meta-model level [22]. His argument is that given a meta-model, it is relatively easy to provide tool support for models created based on the meta-model. Since in our work we are modifying the meta-model for UML by introducing support for crosscutting, it is helpful to know that the question of tool support has been considered by others. We believe that the ability to provide modeling tool support for a modified meta-model, coupled with our meta-model's uniform support for various kinds of crosscutting, supports our view that extending the UML meta-model to include AO concepts should be done from the bottom-up.

Chavez and Lucena also address the issue of extending the UML meta-model to cope with aspect-oriented modeling (AOM) [4]. The meta-model is modified in order to make explicit in UML diagrams what the authors consider to be the main notions in AOP: component (base element), aspect (crosscutting element), join point, crosscutting, and weaving. This proposal has similarities to the composition patterns approach, described in the following section (Section 2.2).

Han et. al. also argue that a formal meta-model will simplify tool support, and so propose a meta-model for AspectJ [8]. Their main argument against heavyweight extensions to the UML meta-model is that they are complex and costly to implement, especially when in this case the authors are interested in building an AspectJ-specific tool, and not a general CASE tool. They start by creating a simplified meta-model for the static structure of Java using the meta-object facility

(MOF), then extend it to include the AOP concepts specific to AspectJ. All the major elements of AspectJ are added as either subclasses of, or associations between, the Java meta-model classes. The authors claim that interoperability with other MOF-based tools for Java and AspectJ is guaranteed, however that would require that everyone adopt the same meta-model for Java and AspectJ.

2.2 Composing Diagrams

This section deals with works which provide some ability to compose diagrams or models which were specified separately, and are possibly incomplete.

Straw et. al. look into composing primary and aspect class diagrams [37]. In their mechanism, conflicts and undesirable emergent properties can be identified either during composition, or during analysis of the composed model. Composition directives can be used to resolve conflicts during composition. However, composition directives require developers to already be aware of the potential conflicts in the model, both within the base and aspect models, which would reduce the ability to develop these models independently. Instead, we believe that a system should help modelers discover this information through simple analysis of crosscutting relationships in the existing model(s).

Clarke and Walker propose the use of composition patterns to specify crosscutting concerns [6]. Composition patterns are based on a combination of UML templates and the merge integration from subject-oriented design [5]. A composition pattern describes the design of a crosscutting requirement independently from any design it may crosscut, and so may be reused wherever it is applicable. By using the parameters in UML templates, which also provide a mechanism for binding the parameters to model elements, they can specify composition of crosscutting behaviour with base designs in a reusable way. This approach requires explicit identification of aspects and binding specifications, and can produce composed diagrams, which show the result of bindings applied to the base design. The main focus of this

approach is on composition during the design phase, in order to validate the design of a composition pattern. However, the authors note that its also possible to maintain the separation through the code phase, using an appropriate implementation model. In particular, the authors consider mappings from composition patterns to AspectJ programming constructs.

Ossher and Tarr present HyperJ [25], a tool which supports multi-dimensional separation of concerns [40]. The main idea behind this approach is that a program can be decomposed in any number of ways. A hyperspace represents the concern space, in the form of a matrix where each axis represents a dimension of concerns, and each point a concern in that dimension. Each partial decomposition, relating to a particular concern, is known as a hyperslice, and hyperslices can be composed into hypermodules using composition rules to take care of any conflicts that may arise. HyperJ is a tool which allows the developer to use these concepts in Java by allowing "identification, encapsulation and integration of multiple dimensions of concerns". They suggest that their approach could be used at any stage of the software development life cycle, but do not describe an implementation of the approach for modeling.

In their Hyper/UML approach, Philippow et. al. create a nice continuation to the previous work, by extending the Hyperspace approach to UML, and use it for the development of product lines [27]. Variability common to product lines is implemented using feature driven decomposition (and composition) according to concerns, which correspond to features. They argue that this may allow a higher degree of automation during development. In addition to UML, models and relations are partly defined by the OCL. Components are modeled using Hyper/UML and are implemented in Hyper/J.

In [7], Georg et. al. develop a two-level structure of composition constraints to deal with conflicts that may occur during composition of aspect and primary models. An aspect model consists of UML template diagrams which describe the

pattern. Each template element specifies properties that will be incorporated into the selected points in the primary model. Composition strategies and composition directives correspond to the two levels of constraints. Composition strategies use high-level heuristics to determine how aspects should be composed with the primary model, while composition directives deal with specific conflicts that arise in the context of particular aspect and primary models.

Katara proposes building a refinement hierarchy for a class or sequence diagram [17]. Each concern is viewed as a collection of superposition steps that define it. The authors use the term aspect to refer to these concerns. All additional functionality crosscuts the starting model, so each is an aspect, even though it is part of the core functionality of the final model. It is possible to merge sequence diagrams to see the composed behaviour of a number of sequences.

Prehofer's work [28] addresses the merging of state chart diagrams in much the same way as we treat sequence diagrams. He aims to show that statechart modeling can be extended to modular composition of features, as well. The behaviour of features is specified individually with incomplete diagrams, which are then composed in a way similar to many of the other approaches described in this section. Hierarchical statechart diagrams (composite states) and parallel composition using concurrent states are used extensively to remove conflicts during composition.

Stein and Hanenberg demonstrate how UML can be extended to show aspect-oriented crosscutting. They use UML collaborations and interactions to specify the details of structural and behavioural crosscutting, respectively, in a given decomposition [35]. UML collaborations are seen as inherently crosscutting elements, since they are only guaranteed to describe the roles that model elements perform in certain situations. Interactions are used because the link used to communicate a message can be seen as the point where crosscutting behaviour can be added. Weaving instructions specify the model elements being crosscut. For structural crosscutting, they define base classes that will be crosscut; for behavioural cross-

cutting, the instructions specify the links in the base collaboration. Their weaving mechanism for UML generates standard UML models from aspect-oriented models. The mechanism's adherence to AspectJ semantics allows for a smoother transition from modeling to development.

In another work, the same authors address the question of the design of crosscutting features in UML, and whether UML has sufficient abstractions for this [36]. The authors focus on graphical representations of the details of crosscutting features, instead of trying to find the best matching representation for crosscutting features on some meta-level. The crosscut and crosscutting elements, the composition strategy, and the join points can all be specified independently. The modeler is required to explicitly state all the crosscutting relationships and join points while designing the system.

Chapter 3

Aspect-Oriented Programming

Aspect-oriented programming (AOP) emerged from the observation that programs often contain concerns which are difficult to fit into any particular modularization. Because of the way modules encapsulate behaviour and structure, there often are concerns that don't belong to a single module, but instead are implemented in multiple locations in different modules. This is similar to modeling, where each diagram can be thought of as a particular module, and the design of a single concept is spread across many modules (diagrams).

AOP focuses on providing support for modular implementation of cross-cutting concerns. In ordinary object-oriented (OO) or procedural programs such crosscutting concerns lead to scattering and tangling in the code. Scattering occurs when the implementation of a single concept is scattered across multiple locations in the code, while tangling implies that the implementations of multiple concerns are interwoven within a single class. AOP enables modular implementations of such concerns. AspectJ [18] is one popular implementation of AOP.

AOP addresses modularization issues by introducing new elements and constructs to the programming language. The main concepts introduced by AspectJ-like languages are advice, aspects, join point model, and inter-type declarations (ITDs).

Inter-type declarations (ITDs) allow for defining fields or operations of a

class, from outside the class. ITDs are placed in an aspect, but define fields or methods of a target class. For example, if method Foo.bar() is defined using an ITD in an aspect, then the method bar() belongs to class Foo, and calls to bar() can be made on objects of type Foo.

Pointcuts are expressions which can pick out points in the execution of a program. Pointcuts are frequently specified using type patterns, which match a particular subset of points in the program's execution. Advice is a mechanism that allows a programmer to modify the behaviour of existing base code, by specifying the code to be executed instead of or in addition to the existing code. Advice can therefore be before, after, or around. Finally, aspects encapsulate advice and ITDs, as well as pointcuts. The pointcut specifies the point(s) where the advice is applied, or the ITDs are introduced. These points are called join points, and are identified within the ontology of a join point model (JPM).

JPMs are the central mechanism that supports crosscutting in AOP [23]. A JPM can be described in terms of three characteristics: the nature of the join points, a means of identifying the join points, and a means of affecting semantics at join points. In AspectJ, dynamic join points are points in the program's execution, they're identified by pointcuts, and the means of semantic effect is for advice to run before, after, or around the join point. In SpringAOP [34], the dynamic join points are method invocations, the means of identification are pointcuts, similar to those in AspectJ, and the means of semantic effect is for advice to run before, after, around the join point, or in case the method throws an exception. Since pointcuts in Spring are simple Java classes, it is possible to declare custom pointcuts simply by writing a new class. In AspectJ the static join points are fields, methods and the parents of a type. With AspectJ, you can add fields, methods, or interfaces to classes, while Spring only allows introduction of interfaces to objects.

AOP implementations such as AspectJ rely on a weaving process to coordinate the execution of advice with join points. In current AspectJ implementations,

advice weaving happens at compile or load time, and is broken down into two stages: planning, and code munging. During the planning stage, join point shadows [11] within the code are analyzed to check whether they match the pointcuts associated with advice. The matching shadows are annotated with each advice that could apply at that point. During the munging stage calls to advice methods are added at the matched shadows. Runtime residual tests can also be added to guard the execution if the advice will match only under certain conditions, which won't be known until runtime.

Chapter 4

Introducing Examples

In this chapter, we introduce the example models we used to test our planner. These will be used in Chapter 5 and Chapter 6 to explain some of the more complicated points of the meta-model and planner, respectively, and also in Chapter 7 to evaluate our framework's ability to simplify the implementation of analysis tools. We will attempt to point out some interesting characteristics of each of the designs, which we will focus on during our analysis.

4.1 Graphical Shapes Editor Example

The model we use is an adaptation of the original graphical shapes example used in the seminal AOP papers [18, 19, 23]. Two model fragments are shown in Figures 4.1 and 4.2. Both contain a class diagram and sequence diagram(s). The top fragment models the main functionality of the Display for Shapes, which include Points and Lines. The bottom fragment models the Subject-Observer design pattern. The bindings of the elements in the top fragment to the elements of the design pattern are included in the upper class diagram. Note that this is not standard UML format for representing collaborations, and we're just using this shorthand representation to simplify the figure, and show the dependencies in a more compact way.

This example is interesting mainly because of the application of a pattern to

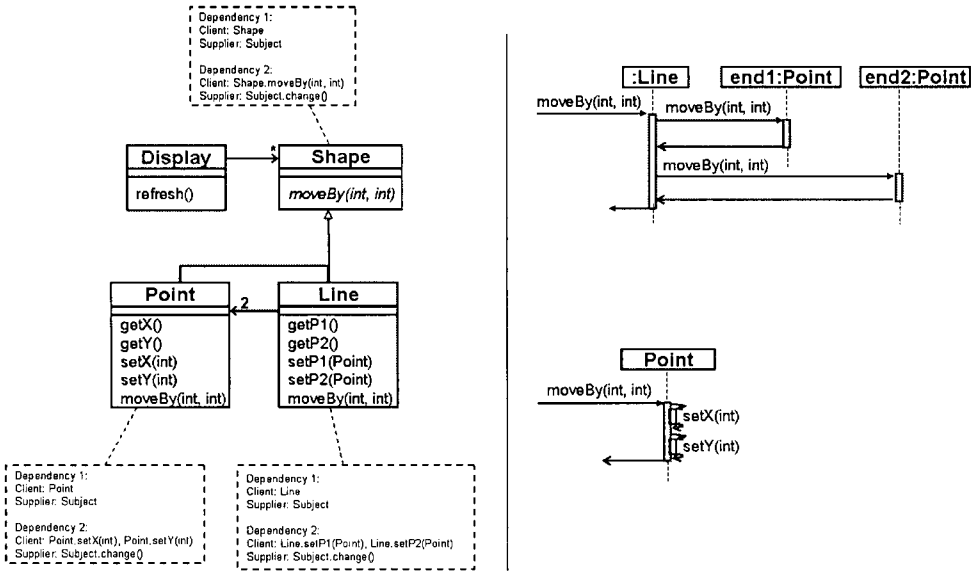


Figure 4.1: Class and sequence diagrams for the Graphical Shapes Editor example.

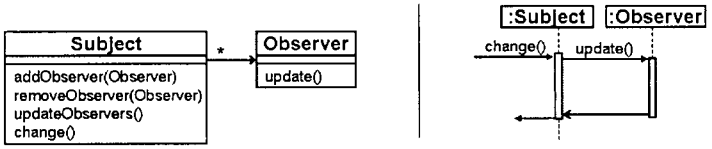


Figure 4.2: Class and sequence diagrams for the Subject-Observer design pattern.

the base design. This will allow us to use the sequence diagram composition tool (Section 7.2) to analyze the behaviour of the final system, which reveals interesting information about the behaviour of the composed system. This is also a classic AOP example, and will be familiar to many of the interested readers when we discuss the possible applications of our JPM and planner to aspect-oriented modeling in Section 7.3.

4.2 RSA Phone Model Example

The second model is more of a real-world example. It is part of a model that we obtained from the Rational Software Architect (RSA) group at IBM-Ottawa, and is one the examples used during their own user testing of RSA product. RSA is an Eclipse-based design and development tool which uses model-driven development with UML to create applications. As such, this is the best example we have of real-world tasks required of users, as compared to the other designs we've used. We used the class diagram and sequence diagrams provided to us as they were. The state machine diagram was slightly too complicated for our purposes, so we created two simple state machines based on the information found in the sequence diagrams.

This example models a simple phone system, where both the Network and the physical components of the Phone (namely Keypad and Display) are represented. A number of different sequence and state machine diagrams specify the behaviour of the system. These diagrams are shown in Figures 4.3 — 4.7.

In this document, we include only two of the three sequence diagrams we used in testing. We do this in order to save space, and also because the interaction sequence for the omitted sequence diagram is just a more detailed version of the first sequence diagram. The first sequence diagram (Figure 4.3) specifies the interactions that take place when a user places a call. The second sequence diagram, one that we have omitted from this document, goes into slightly more detail of the same interaction, including operations like validation. The final sequence diagram (Figures

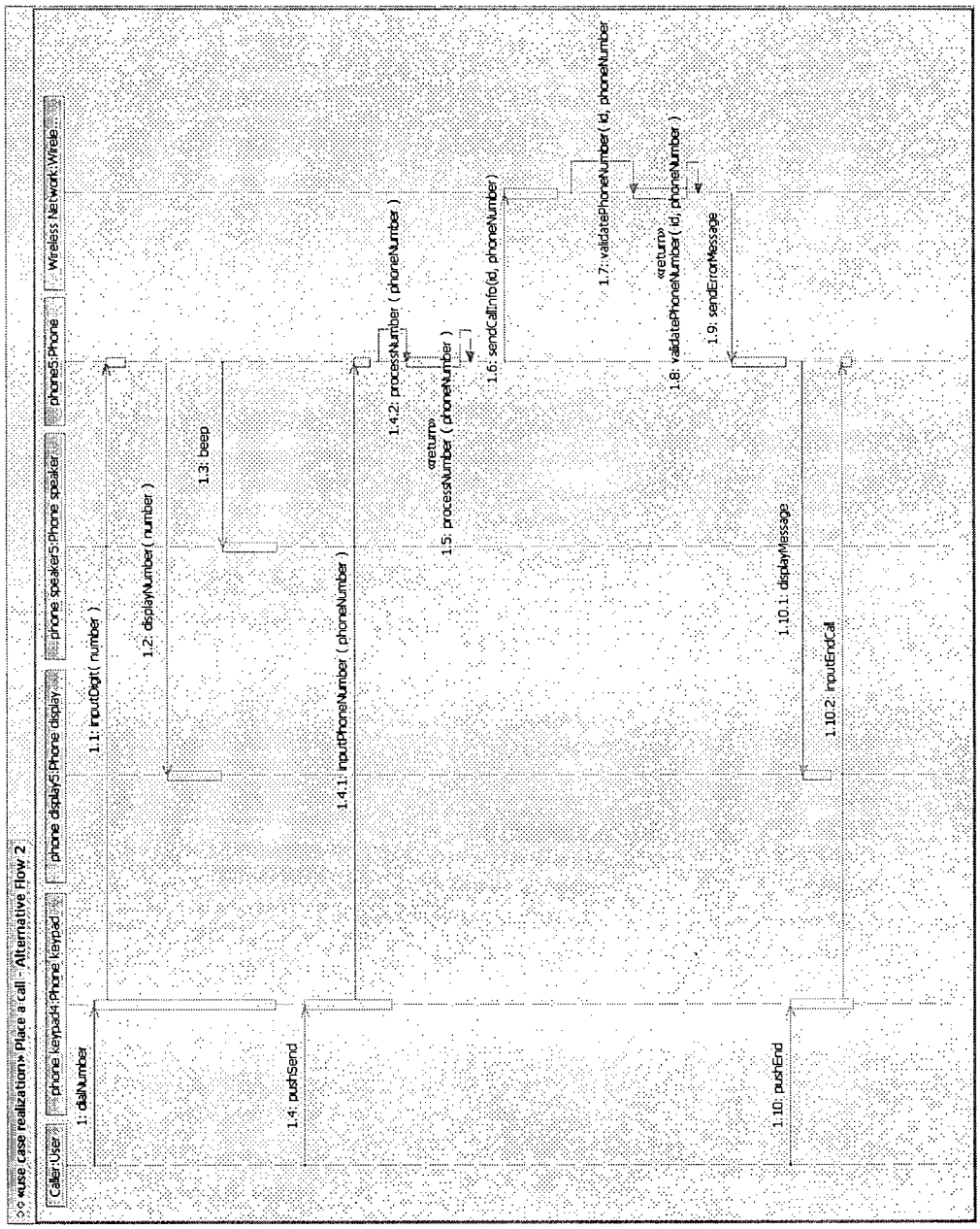
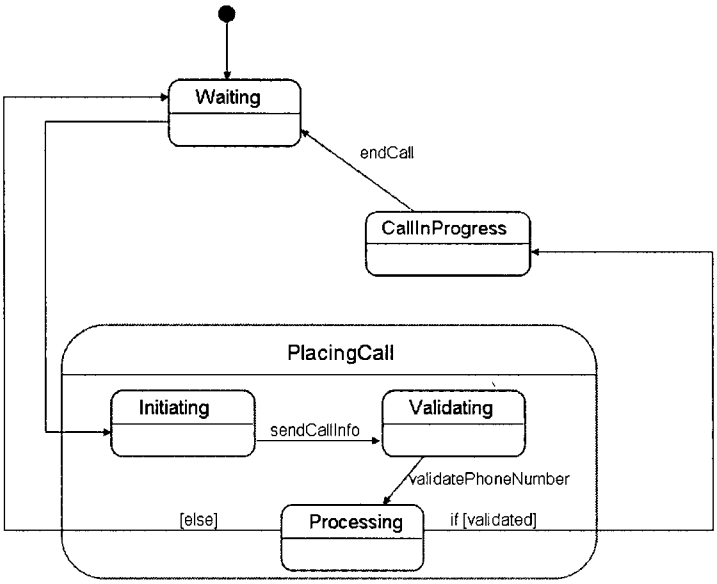
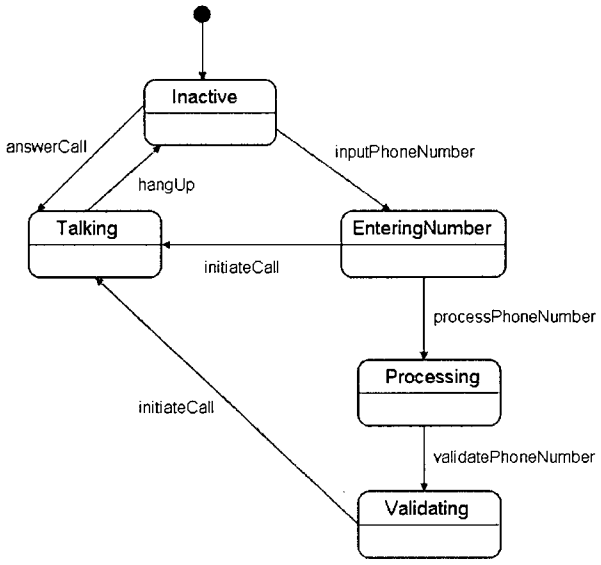


Figure 4.3: Sequence diagram for one user placing a call in the RSA model example.



Network State Machine (NetworkSM)



Phone State Machine (PhoneSM)

Figure 4.4: State machine diagrams for the Network and Phone classes in the RSA example. SM stands for State Machine.

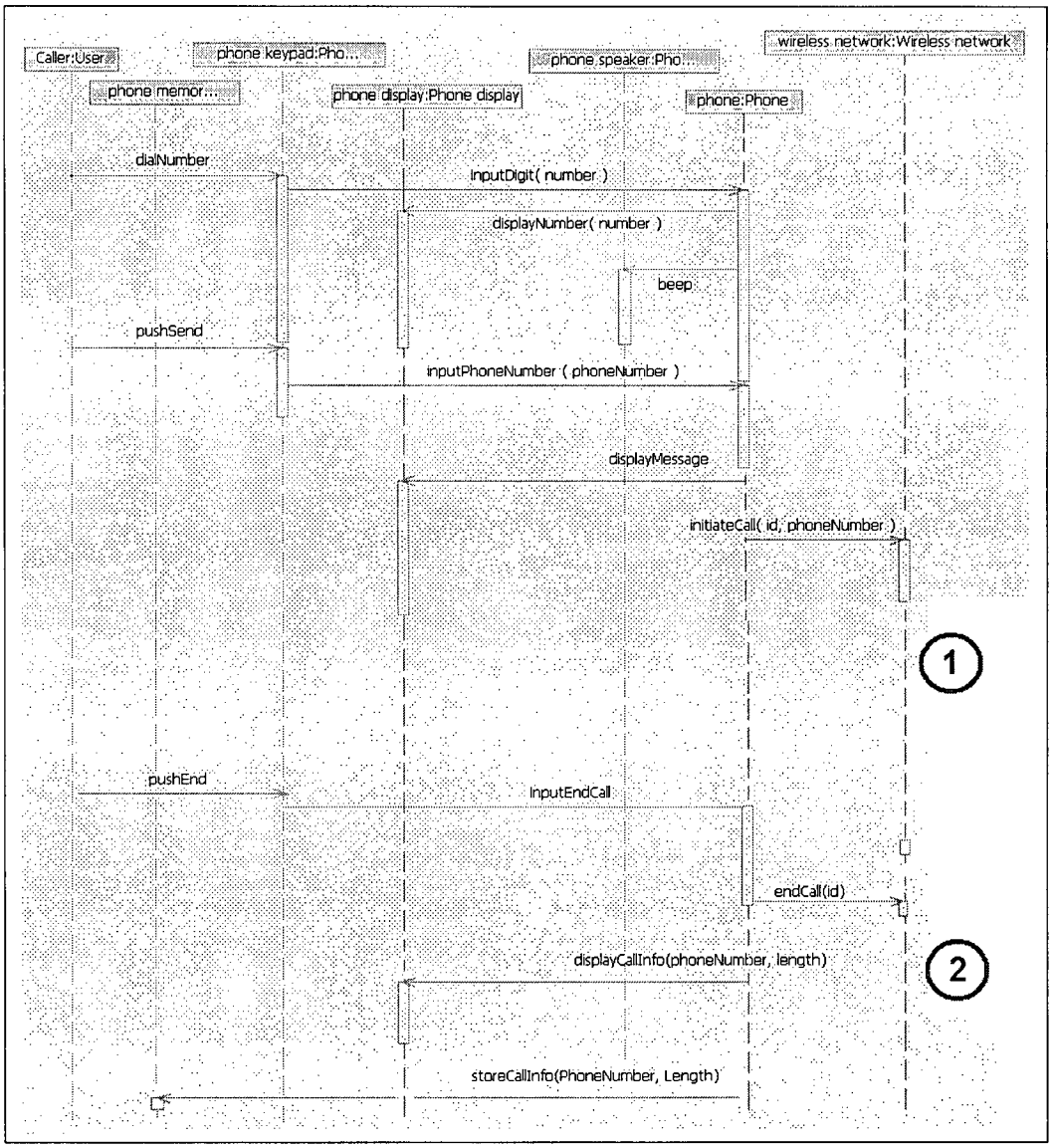


Figure 4.5: Sequence diagram for the two-user call in the RSA example. The diagram is split between this figure and Figure 4.6, with the common link being the WirelessNetwork lifeline. The circled numbers "1" and "2" represent the places where the two diagram fragments link together. The messages following "1" in Figure 4.6 (starting at Phone.checkForIdle) are inserted after the "1" (WirelessNetwork.initiateCall) in the current figure. Same for the circled "2": messages starting at PhoneDisplay.displayCallInfo in Figure 4.6 are inserted after WirelessNetwork.endCall in the current figure.

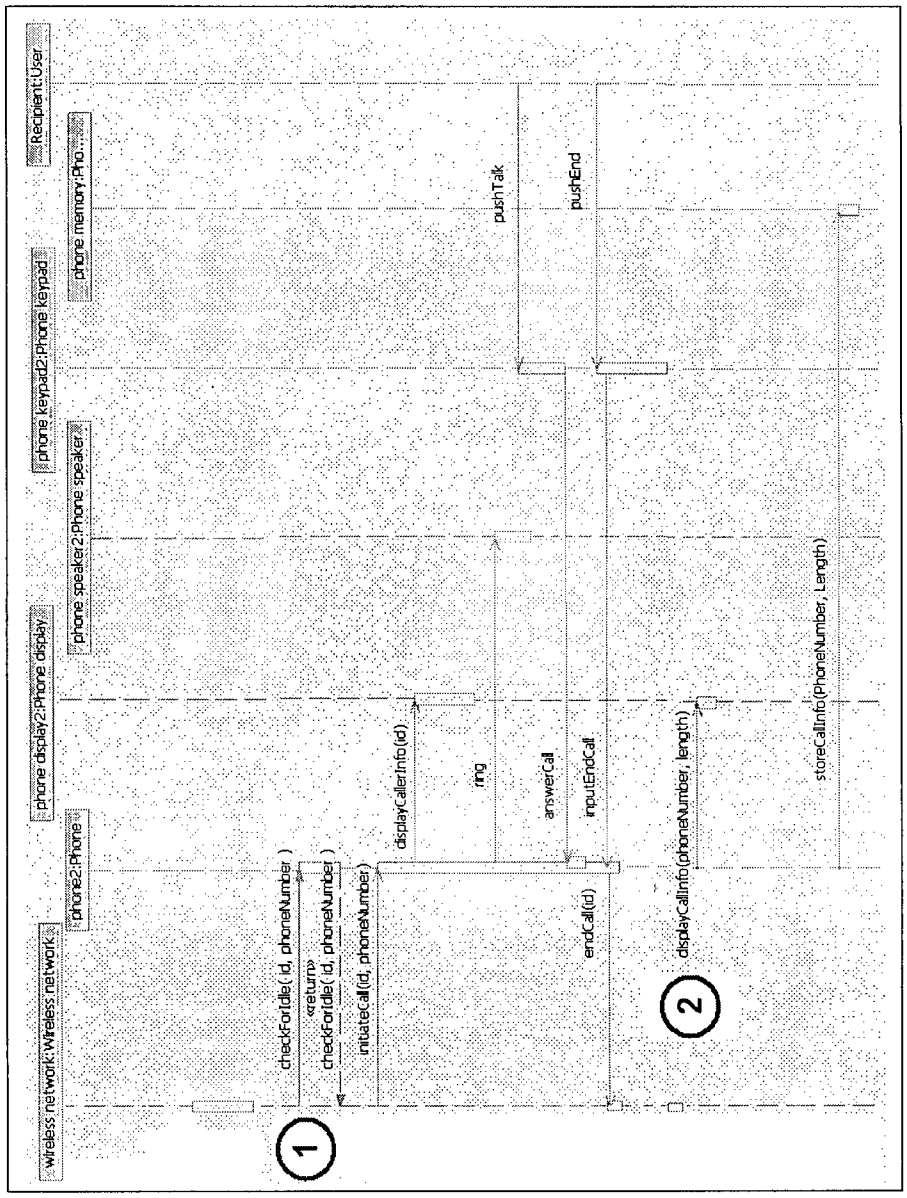


Figure 4.6: Sequence diagram for the two-user call in the RSA example. The diagram is split between this figure and Figure 4.5, with the common link being the WirelessNetwork lifeline. The circled numbers "1" and "2" represent the places where the two diagram fragments link together. The messages following "1" in this figure are inserted after the "1" (WirelessNetwork.initiateCall) in Figure 4.5. Same for the circled "2": messages starting at PhoneDisplay.displayCallInfo in the current figure are inserted after WirelessNetwork.endCall in Figure 4.5.

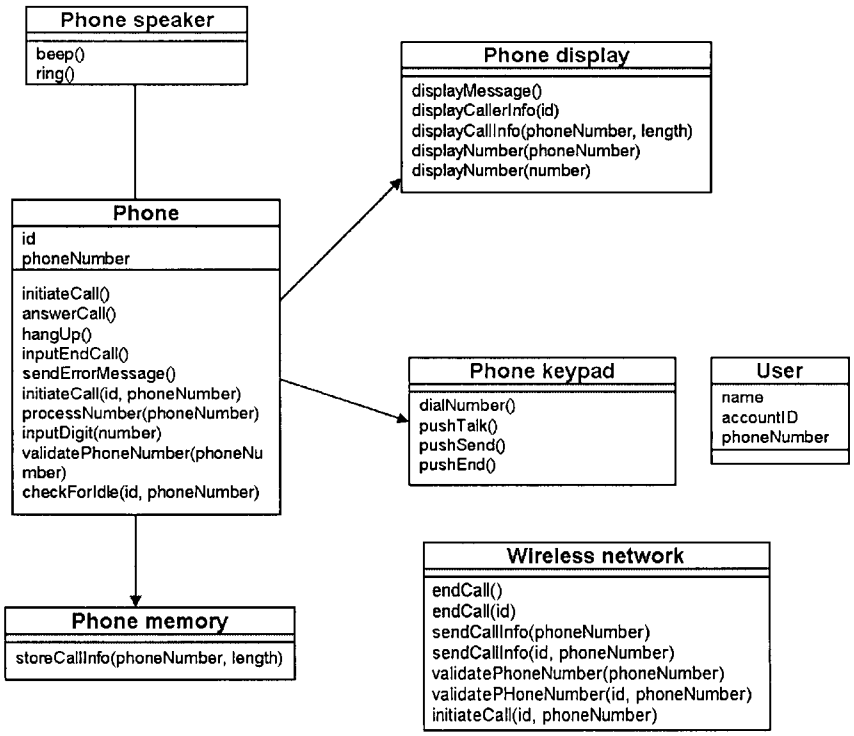


Figure 4.7: Class diagram for the RSA model example.

4.5 and 4.6), shows the specification of the behaviour of both the users involved in a phone call. The sequence is split between the two diagrams, with the common link being the WirelessNetwork lifeline.

The model also contains state machine diagrams for the Phone and PhoneDisplay (Figure 4.4), detailing the states objects of these classes may enter during their use, as well as a class diagram for the system (Figure 4.7).

This model was originally used to test out the recommendation tool for state machine diagrams (discussed in Section 7.1), where, given sequence diagrams and a (possibly incomplete) state machine, the tool recommends which transitions are possible out of each state, based on the messages that can be sent according to the sequence diagram. Since there are a number of reasonably involved sequence diagrams, this example will also be useful for testing the sequence diagram composition tool we describe in Section 7.1.

Chapter 5

Meta-Model Description

We have developed a simple realization of our meta-model on top of the UML2 modeling framework [30], which is an implementation of the UML 2.0 meta-model for the Eclipse platform. It is based on the Eclipse Modeling Framework (EMF) [29]. Our initial implementation actually used EMF, and the meta-model it provides, as a basis for our tool, and we briefly discuss the differences between the two frameworks and the reasons for the switch at the start of the following chapter.

We would like to draw the reader's attention to the distinction between UML2 and UML 2.0. The two terms are very similar, but mean different things. UML 2.0 refers to version 2.0 of the UML language specification, maintained by the OMG. UML2 refers to an EMF-based implementation of the UML 2.0 meta-model for the Eclipse platform. So, UML 2.0 is a *specification* while UML2 is a complete *implementation* of this specification.

Our implementation has three main components. First, we take a subset of the meta-model of the UML modeling language, which includes core elements of class diagrams, sequence diagrams (represented by Interactions), and state machine diagrams. In addition, the class diagrams are extended with a mechanism for inter-type declarations. Second, this subset of the UML meta-model is extended with a JPM. Finally, we present a planner which coordinates crosscutting structure in

models based on this meta-model. The extensions to the meta-model provide the foundation for the planner to record its results. In this chapter, we discuss the first two components. The next chapter deals specifically with the planner implementation.

5.1 UML Meta-Model

A model in our system is formed from any number of class diagrams, sequence diagrams, and state machine diagrams. All of the diagrams are consistent with the UML 2.0 meta-model, since we use the UML2 framework to create our models. Figures 5.1 — 5.4 show the subset of the UML meta-model with which we are working, with the JPM additions highlighted in red and circled. All of these figures are taken from the UML superstructure document, available from the OMG website [24].

The diagrams in UML2 are currently represented in a simple tree format instead of a standard UML graphical notation, since in fact UML2 is a framework that is intended for use as a basis for modeling tool implementation, and not for the creation of models. However, since ours is a proof-of-concept tool, we do not concern ourselves with a proper visual display at the moment. As far as we are aware, there are also no mature projects that would enable us to create a graphical editor/display for an arbitrary meta-model. The GEF project [31] under Eclipse presents one possibility, but it is still in the early stages of development.

5.1.1 Class Diagrams

In class diagrams we support class and interface elements, which can also have properties (fields) and operations. We support generalization relationships, but at the moment we don't support relationships such as dependency, aggregation, realization, etc. We allow operations and properties to be inter-type declarations (ITDs) in that they can be located in one class, but actually define a member

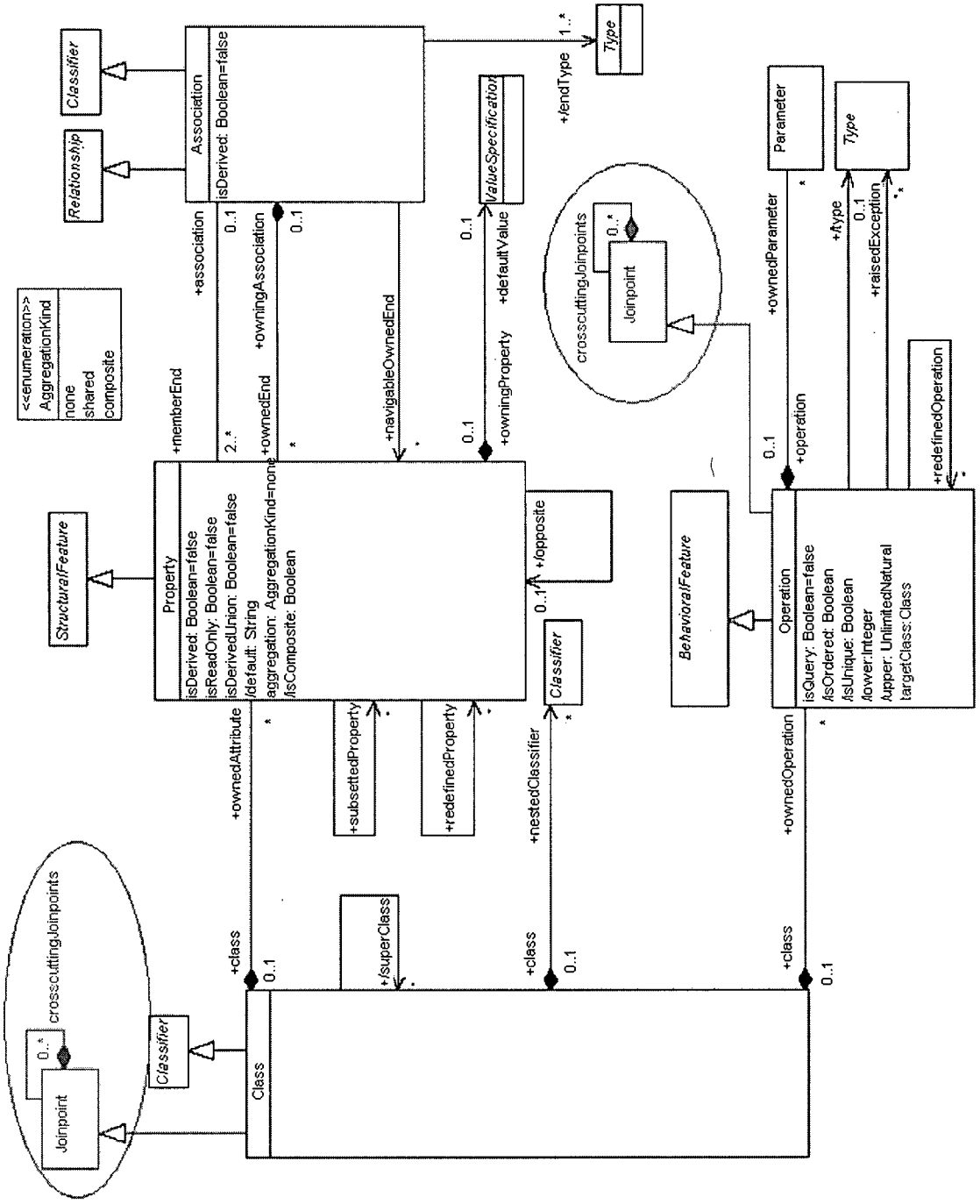


Figure 5.1: Class diagram and collaboration meta-model, from the UML superstructure document.

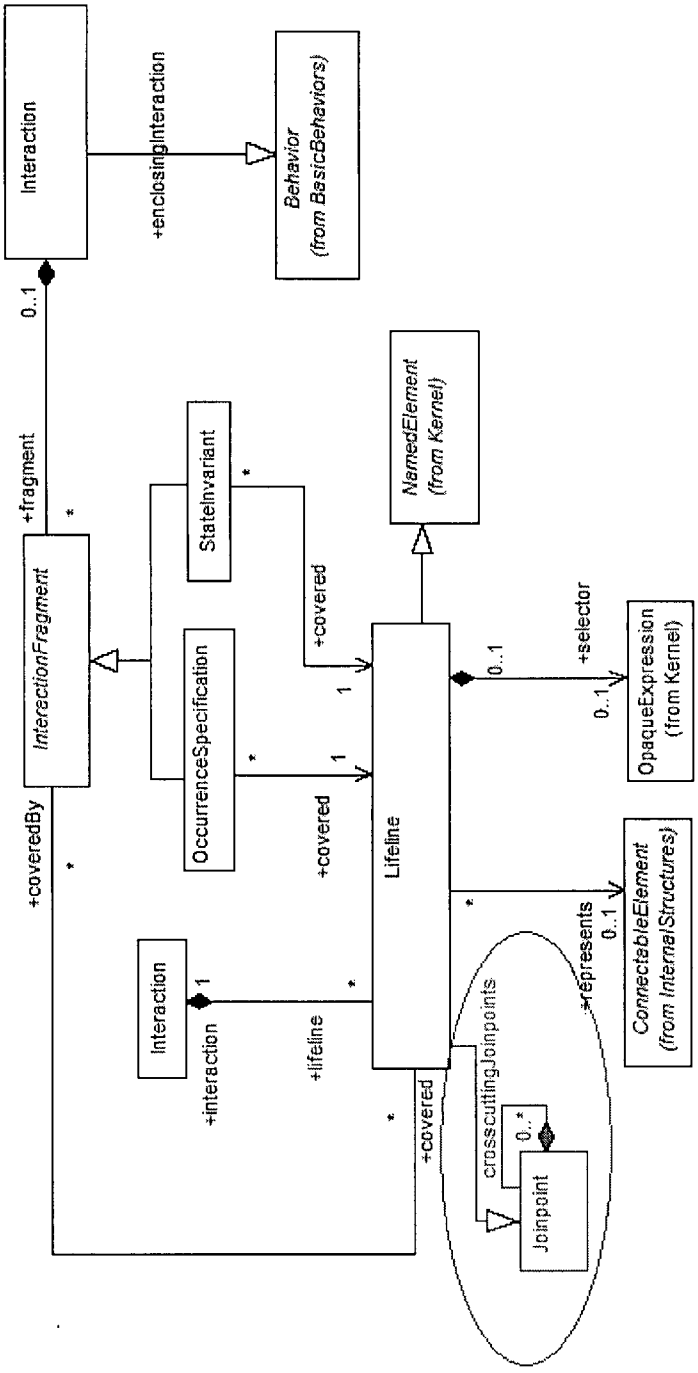


Figure 5.2: Sequence diagram meta-model: lifelines and interactions. Figure 5.3 contains the other half of this meta-model. From the UML superstructure document.

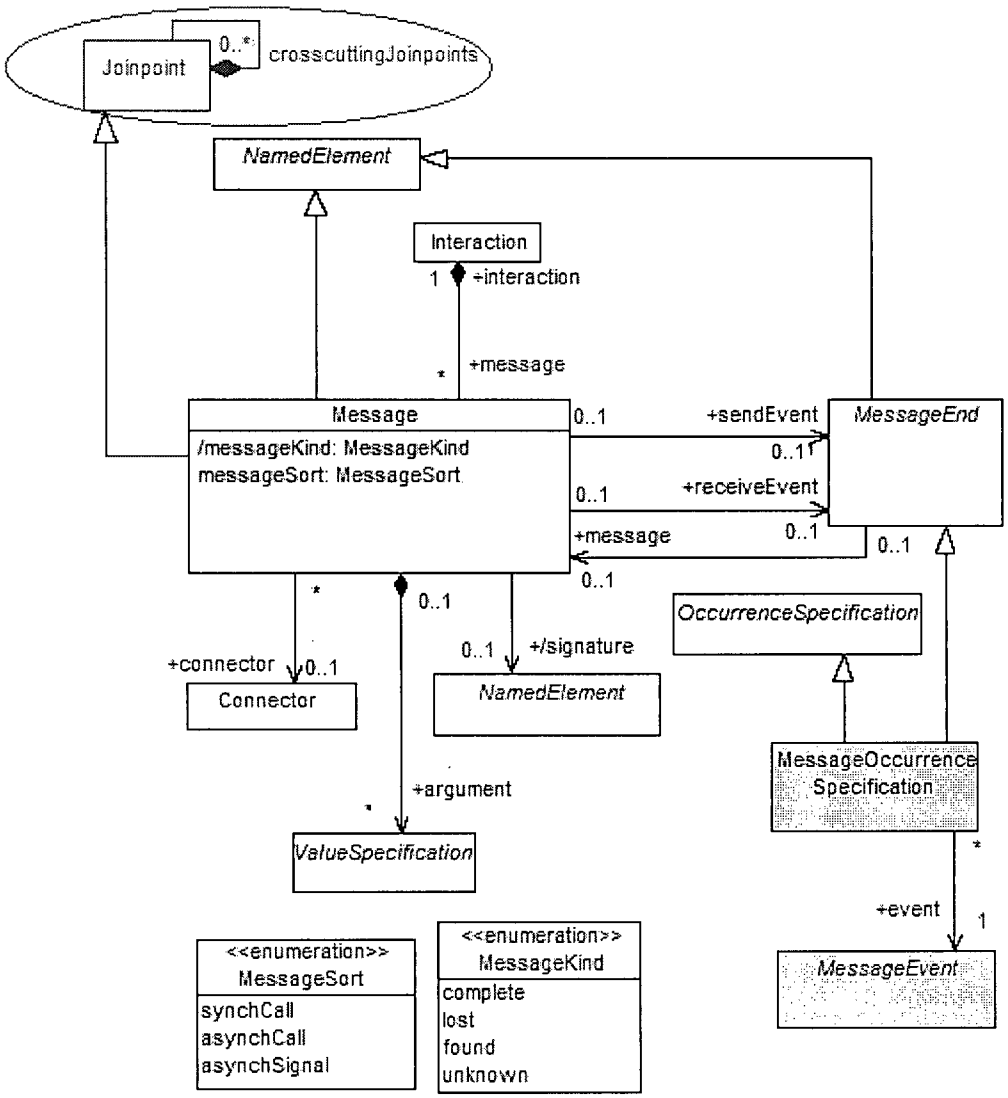


Figure 5.3: Sequence diagram meta-model: messages and interactions. Figure 5.2 contains the other half of this meta-model. From the UML superstructure document.

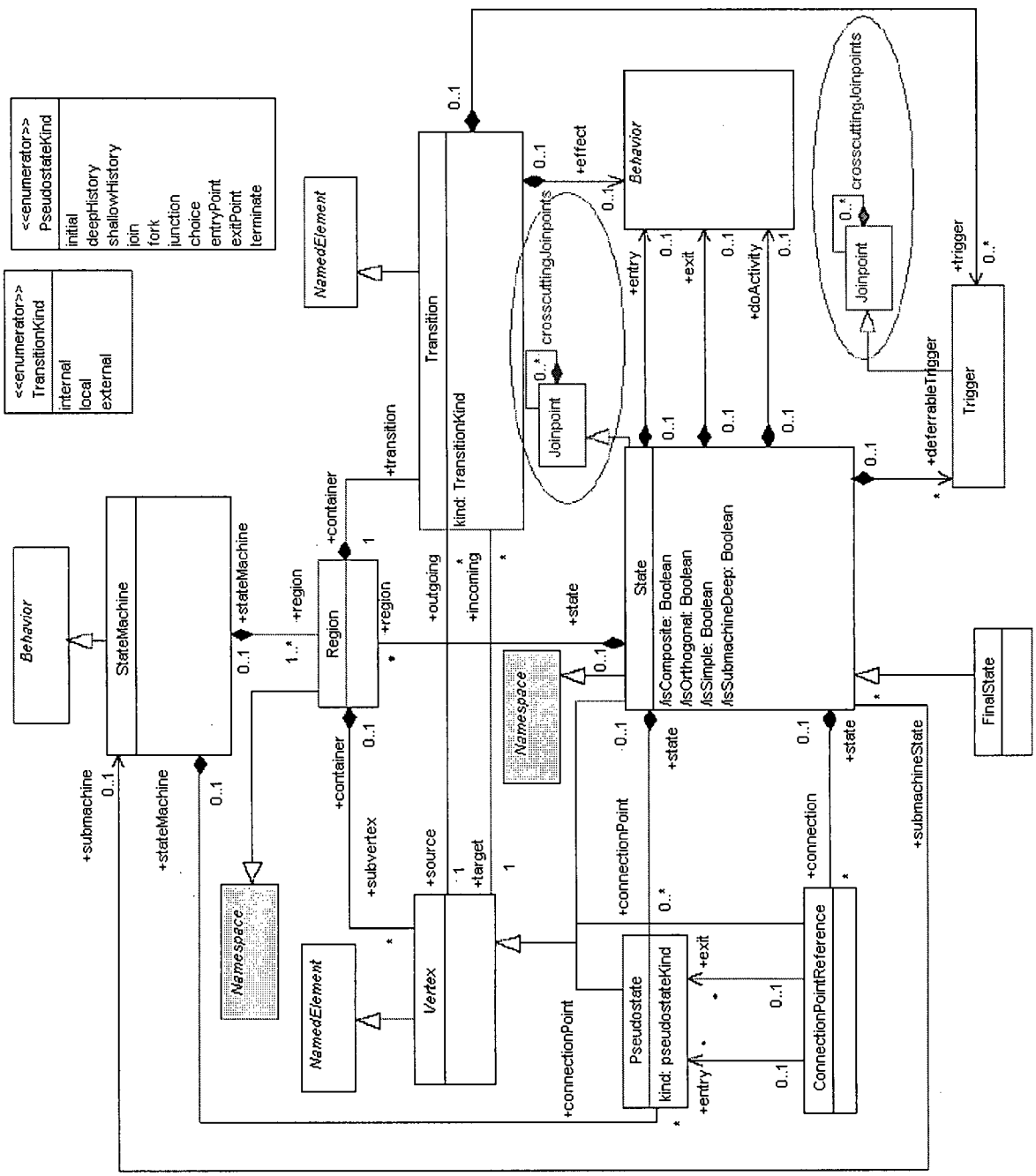


Figure 5.4: State machine diagram meta-model, from the UML superstructure document.

of another class [2, 3], similar in function to ITDs in AOP. This is accomplished through the introduction of a "targetClass" (see Figure 5.1) field to the meta-model elements representing properties and operations. If the "targetClass" field is empty, the element is assumed to be a regular property/operation, and semantically belong to the class in which it is defined. So, unlike AspectJ, there is no special ITD element, but rather each attribute or operation can be either a regular element or an ITD, depending on the value of the "targetClass" field.

Collaborations describe the application of a design pattern to a base model, by declaring the binding between classes or between methods. Each class that performs a role in a pattern declares its own CollaborationOccurrence, which contains a number of Dependency elements. Each Dependency element specifies the binding of either a class or an operation to one of the suppliers of roles (class or operation) in the collaboration. Collaborations also model crosscutting structure, and can be crosscutting in two ways. First, they may refer to elements in two different class diagrams. Second, they may mention elements in different classes of a class diagram.

Figure 5.5 shows the Shapes class diagram from Figures 4.1 and 4.2 as it appears in UML2. We include this figure in order to show the reader the kind of interface available for creating models with UML2.

Initially (in the EMF implementation that will be discussed at the start of the following chapter), we used special role binding elements, which were a new construct we added to the EMF implementation of the partial UML meta-model to represent the collaboration relationships. Role bindings declared the binding of a class to a role or of an operation to a role-operation, with roles modeled as ordinary classes and operations in a separate class diagram. However, with the introduction of collaborations in UML2, this construct became redundant.

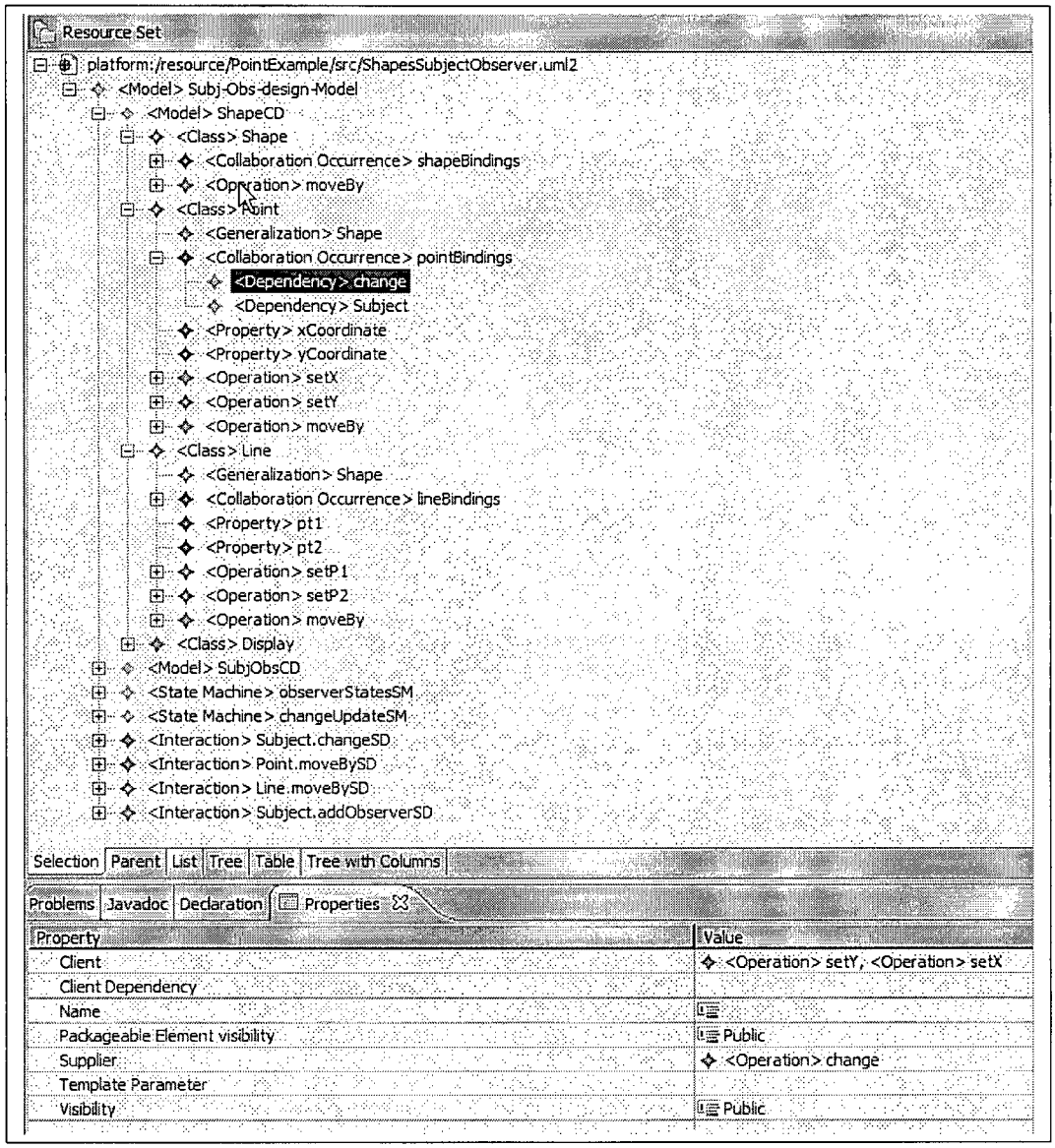


Figure 5.5: Example of a class diagram as shown with UML2.

5.1.2 Sequence Diagrams

A sequence diagram is represented by an Interaction in UML2, where each interaction consists of Lifelines and Messages, as well as EventOccurrence and ExecutionOccurrence elements which are used to determine the ordering of messages in the sequence.

Each ExecutionOccurrence specifies the lifelines that are covered by the execution, as well as the events associated with the sending and receiving of messages at the start and end of the execution. Each Lifeline also keeps track of all the ExecutionOccurrences which cover it, and so all the messages that can be sent to and by the object that this lifeline represents.

Sequence diagrams crosscut class diagrams in that a sequence diagram refers to operations in multiple classes in the corresponding class diagram.

Figure 5.6 shows the Line.moveBy sequence diagram from Figure 4.1 as it appears in UML2.

5.1.3 State Machine Diagrams

A state machine is represented by a StateMachine element in UML2, which is capable of representing orthogonal regions, as well as composite states inside a given state machine. A state machine can be specified by a series of States and Transitions between them, where each transition may be triggered automatically or by a CallTrigger.

State machines model class states and transitions, which crosscut the class and sequence diagrams since they refer to operations/messages on multiple classes/lifelines in the corresponding diagrams.

Figure 5.7 shows what the state machine for the Phone class in Figure 4.7 looks like in UML2. We use this figure to show the presence of the "crosscuttingJoinpoint" field on the CallTrigger element.

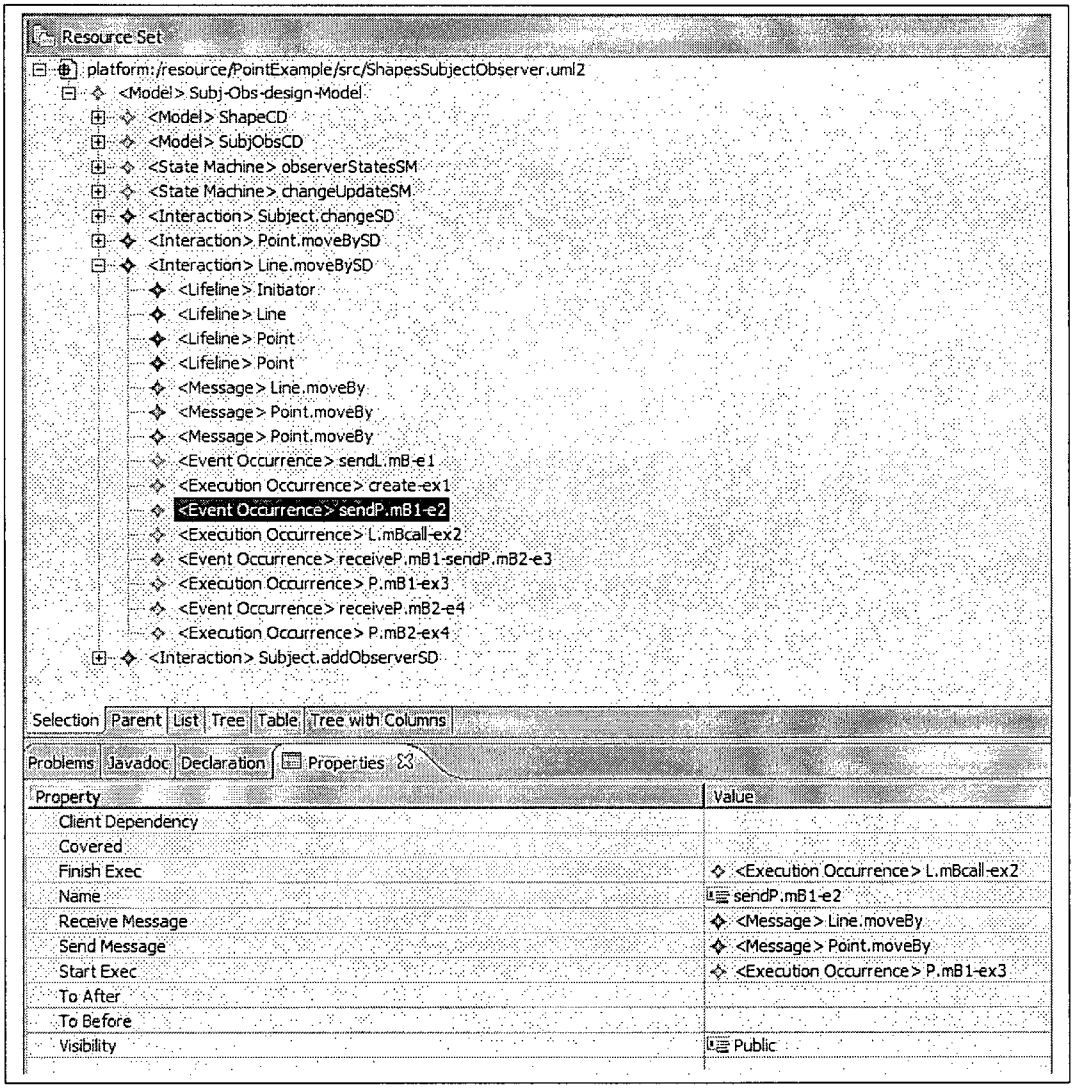


Figure 5.6: Example of a sequence diagram as shown with UML2.

Figure 5.7: Example of a state machine diagram as shown with UML2.

5.2 The Join Point Model (JPM)

JPMs are the central mechanism that supports crosscutting in aspect-oriented programming [23]. A JPM can be described in terms of three characteristics: the nature of the join points, the means of identifying the join points, and the means of semantic effect at join points. Figures 5.1 — 5.4 circle in red the changes we have made to the UML meta-model in order to support our join-point model, and a more detailed discussion of each element in the ontology follows.

5.2.1 Join Points

The join points of our JPM are the selected model elements in the various UML diagrams. Every join point is a model element, but only the elements we mention below are join points.

Each join point also maintains a field, named "crosscuttingJoinpoints", which maintains the crosscut-by set — the list of all the other join points which crosscut this one.

From class diagrams the join points are Class and Operation elements. From sequence diagrams the join points are Messages and Lifelines. From state machines, the join points are States and CallTriggers. Because all of our join points are model elements, we will often refer to a join point as a model element in the following discussion.

5.2.2 Means of Identifying Join Points

Our proposal includes several means of identifying join points. All model elements can be identified either directly by name, or by a compound signature.

The name-based identification is straightforward — elements are matched on their name and also on the name of any of the roles that they perform, as specified through collaborations. For example, an operation has a basic name, which is the name of the operation model element. It also has a signature, which is a combination

of the target class of the operation, and the name of the operation. So, using the Shapes Editor example, for the setX method in the Point class, its name would be "setX", and its signature would be "Point.setX".

The "targetClass" field is used to determine the class that appears in the signature of an operation. Due to the presence of ITDs, the class containing the operation declaration may be different from the class on which the operation is actually implemented. These compound signatures, which can be used to label each of the model elements, are used to match other elements which crosscut it. Equivalent signatures for two different model elements signify that they refer to the same underlying concept, and thus crosscut each other. The matching of signatures is further complicated by the presence of collaborations, and is discussed in detail in Section 6.4.1.

5.2.3 Semantic Effect at Join Points

Our system preserves the original declaration semantics of each model element. Since we are adding a JPM to the existing UML meta-model, where crosscutting is already present (although implicit), our JPM does not define any new semantic effect — it simply accounts for the existing semantic effect within the new framework. Each element records all the other model elements which crosscut it. As mentioned before, this corresponds to the planning stage of an AOP weaver — the information is recorded without any explicit modifications made to the model. Performing the actual weaving would necessitate dealing with the question of how the crosscutting relationships will be displayed, which is beyond the scope of this work. For research that deals more explicitly with the issue of weaving, see [6, 7, 12].

Because our mechanism simply extends the existing meta-model and its semantics, it should be possible to incorporate this proposal into other meta-models, but we have not attempted this yet. This would allow for the automated analysis of crosscutting in models based on any meta-model (e.g. language/area specific model-

Table 5.1: Crosscutting relationships that are recorded by the planner between elements in various UML diagrams.

	Class Diagram			Sequence Diagram		State Machine Diagram	
		Class	Op	Lifeline	Message	State	CallTrigger
CD	Class			x			x
	Op				x		x
SD	Lifeline						x
	Message				x		x
SM	State					x	
	CallTrigger						x

ing languages), as long as the appropriate elements in the meta-model were labeled as join points, and semantics established as to which elements could crosscut which other elements.

5.3 Meta-Model Enhanced With the JPM

Now that we have discussed both the UML meta-model we will use, and the JPM we have designed, we can describe in more detail the crosscutting relationships between model elements that we consider. We will look at each kind of diagram in turn, and for each element in the diagram that is a join point look at the other elements that can be in its crosscut-by set. This discussion is also summarized in Table 5.1.

The main rule we used for establishing the crosscutting relationships to include in the crosscut-by set is as follows: we make the relationship explicit, by adding the appropriate elements to each other's crosscut-by set, if it's not already explicit in the model, and moreover, only if it is a direct relationship. We provide examples of this rule throughout the discussion of each of the diagrams below.

The reason behind this rule is to make sure that the crosscut-by sets do not grow too large. If all the transitive relationships are added to these sets, they will have to somehow be filtered when being displayed, as the amount of information presented directly would be too much for the modeler to take in. The main intent

of the crosscut-by sets was to make the direct crosscutting relationships explicit, while the transitive relationships can be established when needed by following the appropriate crosscut-by sets, instead of doing a global search.

5.3.1 Class Diagrams

Within class diagrams, Operations and Classes are join points.

Operations crosscut Messages in sequence diagrams, and CallTriggers in state machines. These relationships are fairly intuitive — Messages refer to a call to a specific Operation, and CallTriggers are also associated with a single Operation. One might say that Transitions and States, both in state machine diagrams, can also be crosscut by operations and messages. However, we return to our rule in this case: since triggers are already crosscut by operations, and triggers cause a transition, this is no longer a direct relationship. Operations that transitively crosscut a transition can be established by looking at the trigger for the transition, and so don't need to be made explicit in the crosscut-by set. Similar reasoning applies to States, as well.

Classes crosscut Lifelines and CallTriggers. Lifelines in sequence diagrams represent objects of the class' type during a particular interaction sequence, and so belong in this list. Classes crosscut triggers because the execution of an operation on this class can cause the trigger to fire. Transitions and states can also be said to crosscut the class, but again, this relationship can be established transitively through the trigger.

Similarly, one might consider whether Operations and Properties should be in a Class' crosscut-by set, because of possible ITDs. We argue that this relationship is already made explicit through Collaborations and Dependencies, and so falls outside the rule established above.

5.3.2 Sequence Diagrams

Within sequence diagrams, Lifelines and Messages are join points.

Lifelines are crosscut by CallTriggers and Classes. We've already discussed the Lifeline-Class crosscutting relationship. Lifelines and CallTriggers crosscut each other because an event is triggered by a message being sent to the object represented by the lifeline.

Messages can be crosscut by other Messages, as well as Operations and Triggers. Message-Operation crosscutting was already discussed. Messages crosscut other Messages if their compound signatures (discussed in Section 5.2.2) match. This means that Messages crosscut each other either when multiple possible flows of execution are specified in different sequence diagrams, or there are interactions for the role-operations performed by this operation that need to be accounted for. Triggers crosscut Messages in the same basic way they do Operations.

5.3.3 State Machine Diagrams

Within state machines, States and CallTriggers are join points.

In addition to all the other crosscutting mentioned above, States can crosscut other States when the transitions into the states match each other. CallTriggers can crosscut other triggers, when the signatures match. This relationship is useful when establishing the crosscutting between transitions or states.

Chapter 6

Implementation

In order to evaluate our proposal and the ease of exposing different kinds of cross-cutting, we have implemented a simple planner tool. The implementation of the prototype evolved over the course of the project, in order to take advantage of the advances in modeling frameworks, as well as feedback from individuals familiar with the field. Our original implementation relied on the EMF framework, as discussed in Section 6.1. Following this, we introduce the current UML2-based implementation.

6.1 EMF Framework

The original prototype implementation of our planner tool was based on the Eclipse Modeling Framework (EMF) version 2.0.0 [29], which allowed us to create and display class diagrams. Similar to UML2, the drawback of EMF is that it is not designed as a language in which you can create models, but rather as a framework for building tools based on a structured meta-model.

EMF provides a partial implementation of a UML meta-model, stored in an ecore file. This is another tree-based structure that specifies all elements in the meta-model, and their various properties and associations. Through extensions to the EMF implementation of the partial UML meta-model, we added support for simple sequence diagrams, role bindings, inter-type declarations, and advice, as well

as support for the JPM. A more detailed description of the EMF-based JPM and planner can be found in [41].

6.2 UML2 Framework

The current prototype implementation is based on the Eclipse UML2 framework, version 2.0.0 [30]. UML2 doesnt provide direct support for visually creating and editing UML diagrams, its purpose being similar to that of EMF, but it supports all of the semantic elements that might be viewed in any UML diagram. The meta-model is once again stored in an ecore file, and represents all the elements and relationships laid out in the OMG's UML superstructure specification document [24].

With UML2, simple diagrams may be created, viewed and edited only in tree form, which makes understanding the relationships between diagram elements a lot more difficult. However, the models we use for testing are small enough that they are still manageable, even in the tree format. In addition to the model in tree format, we keep as a reference a separate model in standard (graphical) UML notation, which represents the same design. In the graphical model, we can see the relationships between elements more clearly, while the tree format contains all the exposed crosscutting relationships. Then, once the planner populates the crosscut-by sets for all the elements in the UML2-based model, we can look at the graphical model to see whether the relationships shown by the planner exist, as well as check for any relationships the planner may have missed.

UML2 is a great gain over EMF, where only class diagrams were supported easily, and the meta-model had to be extended in order to add support for other kinds of diagrams or role binding. With UML2, sequence diagrams are supported natively, and binding between model elements can be performed through the use of Collaborations. Already, this new implementation has enabled us to explore the crosscutting between two more kinds of diagrams (sequence and state machine), as

well as create a simple implementation of our planner.

In our current UML2-based implementation (our second implementation), we decided to explore the crosscutting between different existing kinds of UML diagrams, instead of re-implementing support for advice. This was motivated by the desire to explore new kinds of crosscutting, as well as to differentiate our work from those that simply add aspect-specific support to UML. In talking to members of the modeling community we often found an initial misunderstanding of our approach, and so wanted to suggest a much wider applicability of our work than simply supporting aspects in UML.

6.3 Model Editor

This section describes the process of the editor implementation. The editor we use to create models based on our modified meta-model is designed as an Eclipse plug-in. We had to do surprisingly little work to obtain a simple editor that would fit our requirements, due to the editor generation facilities provided by UML2.

First of all, we obtained the UML2 plug-in and installed it into our Eclipse development environment. We also checked out the code for the plug-in from the Eclipse CVS, and added it as a project to the environment. That way, we were able to launch a new workbench from within Eclipse, where any changes that we made to the code for the editor would be visible. At the same time, since we had the plug-in installed in our environment, we could use its editing and generation tools when making changes to the meta-model.

We thought reusing the existing UML2 editor implementation would be the most practical since the UML2 editor provided most of the functionality we desired (as described in Chapter 5), and since our changes to the meta-model were limited in scope, we believed that only small modifications to the editor would be necessary.

The code for the plug-in came with a completely specified meta-model (in an ecore file), which we modified as described in Chapter 5. We then used the installed

plug-in's generation facilities to re-generate code for the editor tool from the new meta-model specification, which added the code for the entry and display of the additional model elements. The generation facilities are not complete, and required a couple of additional changes to the code in order to make the new fields show up properly in the editor. However, these were minimal, and we were already familiar with the process from our first (EMF-based) implementation.

After this, launching the project as a separate workbench process in Eclipse allowed us to open an editor that recognized the modified model elements, as well as showed the fields for the crosscut-by sets. After this, we were able to create our test models using the editor, and after running the planner over each model were able to view the resulting crosscut-by sets.

6.4 The Planner

In this section, we describe the implementation of the planner tool. Its implementation is separated into a series of phases, each corresponding to a particular kind of crosscutting. The body of the new planner consists of approximately 1,000 loc in 6 main classes, as well as 4 additional utility classes.

During each phase, the planner records the crosscutting join points in the crosscut-by set of the appropriate element. Before describing the phases in detail, we say a couple of things about finding matching elements in the presence of collaborations and dependencies.

To provide feedback about crosscutting structure, we took advantage of our additions to the meta-model. As mentioned before, in Section 5.2, we added a new field to all join points, which keeps track of all the other join points which crosscut this element (the "crosscuttingJoinpoints" field). The planner adds references to the crosscutting elements to each element's crosscut-by set, and this list is then simply displayed by the UML2 editor, along with all the other properties of the model element.

Currently the planner is not optimized for speed, so we require the user to explicitly trigger it when needed, rather than having it run automatically after every change to the model. If the planner were to run incrementally, we would have to take another approach. One possibility is to take note of the elements that have changed, and only consider those elements when updating the crosscut-by sets in the model. In addition, the appropriate planner passes would have to be repeated for the elements that were added, and elements that were deleted would have to be removed from the affected crosscut-by sets. This problem is similar to the problem of incremental weaving for AspectJ, and incremental model-checking [21], so we hope to be able to apply similar techniques to develop an incremental planner.

6.4.1 Matching

Before describing the planner phases, it will be useful to talk about how the planner identifies matching elements. Section 5.2 mentioned briefly the distinction between pure name-based matching and matching in the presence of collaborations, and we elaborate on it here.

The simplest form of element matching is purely name-based. The signatures (discussed in Section 5.2.2) of the two elements are compared, and if they are equal the elements are said to match.

The other kind of matching happens in the presence of Collaborations. With collaborations, classes and operations can perform the roles specified in the collaboration. In this case, two elements match if one performs the role of the other. Using the Shapes Editor example, any calls to Subject.change will crosscut the operation Point.setX, since setX performs the role of change in the context of the Subject-Observer pattern.

In the current implementation, we rely only on the name of the operation and type of the object when matching signatures. We do not take into account parameter lists or return types of operations.

6.4.2 Phases

Our planner was designed to operate in phases, as we gradually expanded the project from looking at interactions between two kinds of diagrams to three. This also gives us the added benefit of being able to take out some of the phases if we are only interested in exploring a particular kind of crosscutting relationship.

Currently, there are four phases in our planner: role binding (collaborations), ITDs, sequence diagrams, and state machine diagrams, run in that order. In addition, there is a set-up phase that precedes all of these, which simply prepares the models by clearing the crosscut-by sets of all model elements, so that the results from previous runs of the planner don't overlap with the current results. This ordering of the phases also ensures that each new phase builds on the existing structure, recording crosscutting not only between elements in the diagrams it adds, but also going back and recording crosscutting between elements of the new diagram and the existing ones. The results (the crosscut-by sets) are recorded by the planner in the "crosscuttingJoinpoint" field of each model element that is a join point. Figure 6.1 shows a UML class diagram of the main classes and operations in our implementation.

Role Binding (Collaborations)

The role binding phase looks at all the CollaborationOccurrences in the model, and records the associations they specify in internal tables. Specifically, it looks at each of the Dependency elements in the CollaborationOccurrence, each of which specifies a binding between classes or operations.

There are two tables: one for recording bindings for classes, and another for operations. Both of the tables are implemented as a Hashtable that maps a role element (class or operation, in the corresponding tables) to a list of model elements (classes or operations) which perform that role in some collaboration. These tables are then referred to during the following phases, when the tool checks for a

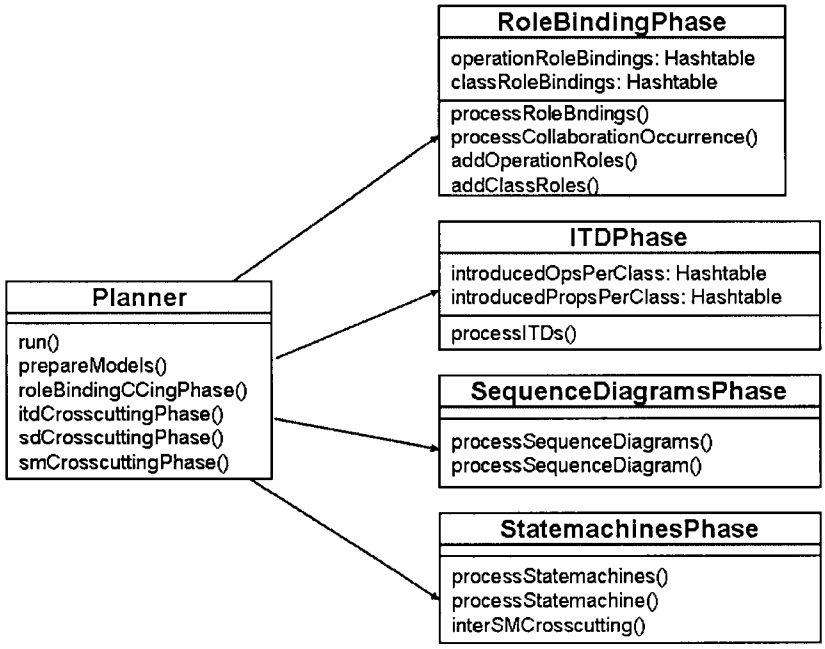


Figure 6.1: Class diagram for the planner implementation, including the major classes and operations. Helper/utility classes and operations are omitted for clarity.

match between two elements, to see whether an element performs some role in a collaboration.

Inter-Type Declarations (ITDs)

The ITD phase goes through the properties and operations in all the classes in the model. It checks the "targetClass" field (described in Chapter 5) of each of these, and if that doesn't match the parent class of the element it records the ITD association in a table. There are two tables in this phase as well, one for recording the introduced operations, and another for introduced properties. Both of the tables are implemented as a Hashtable which maps a class to a list of introduced operations or properties.

The tables from both ITD and role binding phases are used during matching to identify all signatures that could refer to a given model element.

Sequence Diagrams

This next phase goes through the sequence diagrams, and records all the crosscutting that takes place between elements in different sequence diagrams, and between elements in sequence diagrams and class diagrams.

First, the tool goes through all the Messages for each sequence diagram, and finds and records the matching Operations in class diagrams, and matching Messages in other sequence diagrams. Then it iterates through the Lifelines, and finds the matching Classes for the objects represented by the lifelines.

State Machine Diagrams

The final phase goes through the state machine diagrams, and records the crosscutting between elements in different state machines, and also between those in state machines and class and sequence diagrams. It is necessary to go back to look at the class and sequence diagrams at this stage because the information about state

machine elements would not have been available during the previous stages.

For each state machine, the tool first iterates through the CallTriggers, and finds the matching Operations and Messages. Then it finds other matching CallTriggers by looking in all the other state machine diagrams. For all the CallTriggers in the state machines, the planner finds Classes and Lifelines that crosscut them (for a state which is entered after the trigger is set off, this set could be calculated as a combination of the sets of all the triggers for its incoming transitions). Finally, this stage is completed by finding States in different state machine diagrams that crosscut each other. Two states crosscut each other if any of the incoming transitions for both states share any operations in their crosscut-by sets. In other words, this happens when two transitions happen on a call to the same operation, i.e. the transitions crosscut each other.

Chapter 7

Evaluation

To validate our contributions, we use our JPM-enhanced meta-model and planner in the implementation of a couple of simple automated analysis tools. In this chapter, we discuss two such tools, providing examples of their use as well as a discussion of their implementation. We point out a couple of the more interesting crosscutting relationships that our tools uncovered in the examples. Finally, we present a way to add support for advice to the JPM and planner, along with a discussion of the differences of advice in our system and in standard AOP implementations like AspectJ. We have not implemented this addition due to time constraints, but a simple version of it was implemented for our original EMF-based meta-model and planner. This, coupled with the fact that there are no major conceptual differences between the EMF-based and UML2-based approaches, leads us to believe that implementation of such a tool for the current framework would be fairly straightforward.

We argue that analysis or display tools using our framework will be easier to implement, and will be able to present the modeler with more information about the model than RSA [13] or EMF [29] provide. Once the crosscutting relationships are made explicit by the planner, there are any number of ways in which these can be displayed to the modeler. They can also be used as input to automated analysis tools that check for consistency or completeness of the model.

In terms of our contributions, we would like to explicitly outline how each of these was achieved, and what benefits were obtained from accomplishing each task. We highlight some of the benefits during the discussion of each of the problems, and summarize the main points at the end of this chapter.

7.1 Recommending Transitions for State Machines

When designing a system, it is often necessary to be able to say which diagrams can/need to be specified before others. Since multiple diagrams in UML can be used to specify similar things — for example, behaviour for sequence and state machine diagrams — it is important to keep the behaviour specified by both kinds of these diagrams consistent throughout the model. The need for such an application was brought to our attention through talking to some of the people doing usability testing on RSA, whose customers had told them that it would be beneficial to see the transitions available out of a given state in a state machine, based on the interactions already specified in sequence diagrams. In general, we believe this kind of analysis would be useful to those specifying the behaviour of a system, keeping efficiency and consistency of the model in mind.

7.1.1 Solution

This first analysis tool is used to help with the creation of state machines, provided all the sequence diagrams in the model have already been specified. For a selected state, we'd like the tool to be able to make suggestions for possible outgoing transitions based on the sequence diagrams already specified in the model. This feature was implemented as follows: for each State that we would like recommendations for, we looked at the CallTriggers for each of the transitions coming into the state. We then looked at the crosscut-by sets of each of the CallTriggers, and picked out only those join points that were typed as Messages in a sequence diagram. Then we looked at the parent Interaction for each of those messages, m, to see what other

messages could occur in sequence after m in each of those sequence diagrams. Those messages are the ones whose corresponding operations could trigger a transition out of the state. This simple tool can be written with approximately 40 loc, on top of the planner implementation discussed in Chapter 6.

One interesting question is whether to use the message which immediately follows m, or to look a couple of messages down the interaction sequence. One idea we have is to look at the next message to the same object (lifeline) as m, instead of the next message in sequence. It would be necessary to talk to modelers directly, or run experiments on more sample models, to find out which approach would be more useful, before choosing the final implementation. For now, we chose to use the next message in sequence.

An extension of this approach would be to look at states in other state machines which crosscut this state, and recommend messages for the outgoing transitions from those states, as well. We did not explore this approach because the problem as presented to us was how to provide help in creating state machine diagrams when some sequence diagrams were available, without knowing anything about other state machine diagrams.

A small problem arises when looking at the initial message in a sequence diagram. The recommendation tool is currently not complete in that it could never recommend the first message in any sequence diagram, since it is never preceded by other messages that would be picked out from the crosscut-by set of the CallTrigger. We have one suggestion for dealing with this, but it would not be as precise as the other recommendations made by the tool. The basic idea is to look at the signature of a message that is first in a sequence, and determine what class the corresponding operation belongs to. Then, to check whether the class crosscuts the state we're making recommendations for — if so, then we can add the message to the list of recommended messages.

```
file:/C:/Temp/test-workspace/PointExample/src
Working with model: RSAPhoneModel

Working with SM: PhoneSM
...
Outgoing suggestions for state Talking
      Current trigger is initiateCall
            Matching message PhoneDisplay.displayCallerInfo
      Current trigger is initiateCall
            Matching message PhoneDisplay.displayCallerInfo
      Current trigger is answerCall
            Matching message PhoneKeypad.pushEnd
Outgoing suggestions for state EnteringPhoneNumber
      Current trigger is inputPhoneNumber
            Matching message PhoneDisplay.displayMessage
            Matching message Phone.processNumber
            Matching message PhoneDisplay.displayMessage
...

Working with SM: PhoneDisplaySM
...
Outgoing suggestions for state displayingCallerInfo
      Current trigger is displayCallerInfo
            Matching message PhoneSpeaker.ring

...
Done.
```

Figure 7.1: Snippet of the output of the transition recommendation tool for state machines, as run on the RSA model example.

7.1.2 What We Found in Examples

In this section, we show the results of running the tool on the RSA example model. Figure 7.1 shows part of the output of the tool. Only the signatures of recommended messages are shown, but the tool gets a reference to the message itself. This would be useful if the analysis tool were extended with a graphical interface, and then the tool would be able to show direct links to the recommended messages, or even create the recommended transitions automatically.

For the state EnteringPhoneNumber, in the PhoneSM (state machine), the sequence diagrams recommend that transitions out of the state could happen on calls to PhoneDisplay.display and Phone.processNumber. The second transition already appears in the PhoneSM, which is a good check to make sure that the sequence diagram and state machine are in agreement. If the state machine hadn't been

complete, we would know that at least one of the recommendations made by the tool was the one that was chosen by the designers.

For the state Talking, also in the PhoneSM state machine diagram, PhoneDisplay.displayCallerInfo can cause a transition out of the state, since the user may want to know who they're talking to. Also, PhoneKeypad.pushEnd can cause a transition, which makes sense since a conversation needs to be ended after it's started, and the caller can do this by pushing the "end call" button on their phone.

7.1.3 Benefit

The JPM and planner have simplified the implementation of this analysis tool by exposing the crosscutting relationships we are interested in, and collecting them in an easily-accessible list.

We anticipate benefits from such an analysis tool itself, but those are outside the actual scope of this project. For example, the design of state machines will be easier and less error-prone if there is already information available about transitions possible out of each state when they are being created. On the basis of this, it will also be easier to check for consistency between state and sequence diagrams. In fact, this analysis tool is already a step towards that – transitions based on information from sequence diagrams are recommended, but their use is not enforced.

7.2 Composing Sequence Diagrams

Another challenge when working with behaviour in design models comes from using design patterns. Ideally, the design pattern and its behaviour are specified separately from the base design, or even a design pattern model supplied by someone else is reused. The design pattern is "applied" to the base design through the use of collaborations, which mark operations in the base design as performing roles (of classes or operations) in the pattern model. The challenge comes when trying to understand the composed behaviour of the system. A number of papers have

addressed this issue already [17, 35], and we discuss here how our approach can also be used to deal with this situation.

In a more general sense, there are other cases when you want to compose multiple sequence diagrams, and see the resulting behaviour. One such instance is when there are multiple possible flows of execution for the same operation, and you want to see all the flows that the call to this operation can invoke.

This kind of analysis tool would be useful to anybody wanting to verify the behaviour of a system, and whether it behaves as expected in the presence of roles. In the case of collaborations, this would probably be the person adding the pattern implementation to the base design. Also, the second scenario might be interesting to anybody trying to find out, for example, how to implement a particular operation — they would need to see all the possible executions in order to understand exactly what the operation is responsible for.

7.2.1 Solution

This second kind of analysis tool can help a modeler with overall understanding of the behaviour of a system, where different parts of the behaviour are specified in different sequence diagrams. We propose an analysis tool that will compose two crosscutting sequence diagrams, and display the resulting sequence diagram. There are two major cases where we think this kind of analysis would be useful.

The simplest case is when there are multiple sequence diagrams that share some calls, and we would like to see a composed sequence diagram. For example, in the Shapes Editor Example (Section 4.1), Figure 4.1 shows two sequence diagrams, both of which include a call to Point.moveBy. What would the complete execution look like if Line.moveBy was called? Another variant of this case is where the sequence diagrams specify different alternative executions. In this case, it would still be useful to see the composed diagram, where all possible flows of execution would be indicated.

The other case deals with behaviour in the presence of collaborations, where a role performed by a class or operation can introduce new behaviour. In particular, we are interested in operations that have their execution specified by a sequence diagram, and are also covered by a collaboration, with a separate sequence diagram specifying behaviour for the role-operation. In this case, it would be beneficial to see the composed sequence diagram which includes both of these behaviours, in order to detect any unexpected interactions that the collaboration may introduce.

Sequence diagram composition for both of these approaches can be accomplished by finding all messages in the sequence diagram of interest that are crosscut by any message, m, in some other sequence diagram, and inserting the sequence of messages following m into the original sequence. For the first case, we are only interested in looking at crosscutting messages that have the same signature as the message of interest. For the second case, we are interested in looking at crosscutting messages with a different signature, which will be the messages for the role-operation the corresponding operation may perform. Since roles are taken into account when creating the crosscut-by sets, role-messages will be present in the crosscut-by set of the message of interest. This tool can be written with approximately 50 loc, on top of the planner implementation.

7.2.2 What We Found in Examples

In this section, we show the results of running the analysis tool on the Graphical Editor Example model. We present two examples which correspond to the two cases we introduced in Section 7.2.1.

Figure 7.2 shows output from running the tool on the first example, where we are interested in seeing the full execution flow of the Line.moveBy sequence, without looking at roles.

Figure 7.3 shows output from running the second example, where we want to see how the Point.moveBy sequence is affected in the presence of the change role

```
Working with model: Subj-Obs-design-Model
  Line.moveBy : from Line.moveBySD
    Point.moveBy : from Line.moveBySD
      Point.setX : from Point.moveBySD
        Point.setY : from Point.moveBySD
      Point.moveBy : from Line.moveBySD
        Point.setX : from Point.moveBySD
          Point.setY : from Point.moveBySD
        Point.moveBy : from Line.moveBySD

Done.
```

Figure 7.2: Output of the sequence composition tool on the Line.moveBy sequence from Figure 4.1, with role bindings not included.

```
Working with model: Subj-Obs-design-Model
  Point.moveBy : from Point.moveBySD
    Observer.update : from Subject.changeSD
    Point.setX : from Point.moveBySD
      Observer.update : from Subject.changeSD
      Point.setY : from Point.moveBySD
        Observer.update : from Subject.changeSD

Done.
```

Figure 7.3: Output of the sequence composition tool on the Point.moveBy sequence from Figure 4.1, including role bindings.

on the moveBy, setX, and setY operations of Point.

In particular, we can see that multiple calls would happen to Display.update if all of these operations perform the change role. This is obviously undesirable — in fact, we would like an update to happen only once for each sequence that involves one or more changes. In this case, the tool did a good job of pointing out a possible problem with the design, which can be fixed during implementation by using AOP techniques [18].

7.2.3 Benefit

We believe this kind of analysis should help verify composed system behaviour in the presence of design patterns or advice (discussed in Section 7.3). For example,

with respect to design patterns implemented as collaborations, this could be used to check whether the role behaviour applies in all the places where you would expect it to apply, or whether some operations were missed when dependencies were being specified.

Implementers could also use this approach to figure out exactly where the control flow could go from a given operation, and know exactly which functionality the operation is responsible for. This could also help them catch any discrepancies between design and specifications/headers for operations given to them.

7.3 Advice

This section looks at adding support for new crosscutting elements, both to the meta-model and planner. We consider advice (from AOP) as another kind of modeling element, which can crosscut other (existing) model elements. In particular, advice can be crosscutting in that it may apply to multiple elements in different diagrams in the model. For an introduction to advice and other AOP concepts, please see Chapter 3.

This discussion is more abstract than the previous two, since we have not actually implemented this problem in our UML2-based meta-model and planner. However, we did have support for this in the original EMF-based version, and were able to implement a couple of simple analysis tools on the basis of it [41].

In addition to describing how support for advice can be added to our meta-model and planner, we consider how the advice elements we propose are different from the concept of advice in AOP languages such as AspectJ. In particular, we noticed that advice in AspectJ can only be applied to methods and calls, whereas with our meta-model and JPM advice can be applied to any element in the model that is identified as a join point within the JPM. Of course, this also involves coming up with a more general definition for what it means for an element to be advised.

The first part of this discussion should be of particular interest to those inter-

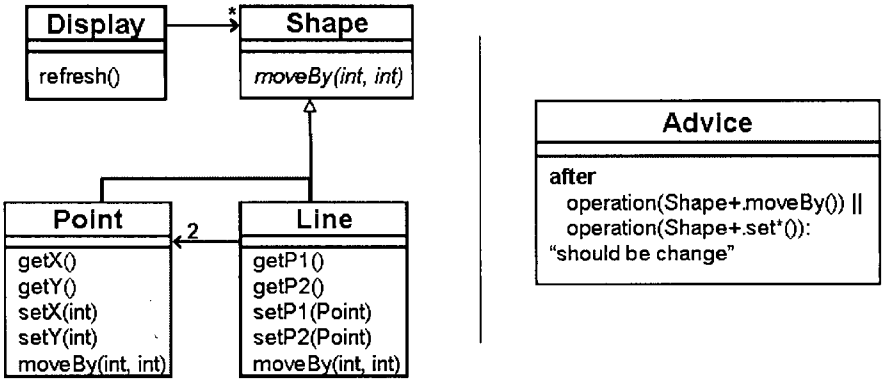


Figure 7.4: Graphical Shapes Editor example class diagram, with an example of advice.

ested in trying out modeling of new paradigms, not just specifically AOP. Because if we can show that support for any kind of element can be added to the meta-model and the planner, people may be more willing to try to import new paradigms from programming to modeling. The second part of this discussion will be more appealing to those interested in exploring the nature of advice in aspect-oriented systems.

7.3.1 Solution

As mentioned before, three changes need to be made: to the meta-model, the JPM, and the planner. First, we add the new advice [18] element to the meta-model, which makes it possible to advise other model elements. Advice in this context simply means that there is some sort of a note attached to the advised element, and it contains the body of the advice. An example advice is shown on the right-hand side in Figure 7.4. The syntax used in the figure serves only to illustrate the meaning, and is not a concrete proposal for advice syntax.

Advice can be crosscutting in that it may apply to multiple elements in different diagrams in the model. Advice can be of three different kinds — before, after, and around. For example, the moveBy and setter methods of Point and Line can be advised with after-advice that makes calls to Display.update. A more

thorough discussion of this example can be found in [41].

In addition to advice elements, we also need to add support for pointcut elements, which specify where the advice can apply. There are many different kinds of pointcuts already present in AOP systems, and some would have to be added in order to pick out each of the model elements identified as a join point. But for the purpose of this example, we are only interested in the operation pointcut (methods are called operations in UML diagrams). The operation pointcut is pretty straightforward — it picks out operations which match the signature specified in the pointcut. We use AspectJ-like syntax in the pointcut expressions. The symbol "+" refers to subclasses, and "*" is used as a wildcard. So, for example, the first pointcut specifies all moveBy operations in subclasses of Shape, while the second refers to all operations in subclasses of Shape whose name starts with "set". Pointcuts that can be used to include/exclude messages in certain cases (like cflow) provide much more finer-grained control over where advice can apply than can be accomplished with collaborations.

Second, we update the JPM — the definition of join points needs to be expanded to include the new elements, although the means of identifying join points and the semantic effect should remain the same.

Third, we need to modify the planner. This will involve either adding a new phase, if we're dealing with a completely new kind of crosscutting, or modifying an existing phase, if we're adding an element in a diagram for which we already provide some support. In general, the addition of a new kind of crosscutting model element requires a new phase of processing if the element is not subsumed by any of the phases described in Chapter 6. In addition, the position of the new phase in the ordering has to be determined. None of the other existing phases should need to be modified to accommodate this change. In this case, we will need to add a new phase to handle the processing of advice, since it's a completely different kind of crosscutting.

The presence of advice in our modeling language, in addition to collaborations and inter-type declarations (ITDs), leads to an interesting semantic question. We would like ITDs and advice to be able to depend on role bindings when appropriate. Specifically, we would like an ITD onto a role element to have the same effect on classes performing the role as members defined directly in the role. Similarly, we would like pointcuts to be able to depend on collaborations and ITDs.

This semantics is easy to achieve with a simple linear processing of model elements in which collaborations are handled before ITDs, which are handled before advice. This would mean that the advice phase would run after all the existing phases. But if we also wanted role bindings to be able to depend on ITDs, then we would have to adopt some sort of a fixed-point approach in our planner. So far, we have been unable to come up with a sufficiently compelling example that would require the more complex semantics.

7.3.2 Benefit

The addition of support for a new kind of crosscutting element to the meta-model and planner has a number of benefits. First, most directly, it will benefit those trying to do aspect-oriented modeling in UML. AOP is in increasing demand right now, and there is a need for support for the process throughout its lifecycle. There are a number of approaches, discussed in Chapter 2, that advocate either the support for AO elements in modeling, or support for AO development, from design through to implementation.

A second, more indirect benefit of making these changes is to show that support for any new kind of crosscutting element can be added to the meta-model, as well as the planner. Since the treatment of all the crosscutting elements is generic, we can see that the implementation of the planner doesn't rely on the kind of element being added, only on the crosscutting relationships it has with other model elements. This can be done for other diagrams in UML, or domain-specific models for certain

applications. This also supports our decision to separate the planner into phases, as the addition of support for new crosscutting elements makes the least possible impact on existing phases.

Because we layer a join point model, which is an aspect-oriented concept, on top of the UML meta-model, instead of simply adding direct support for AspectJ elements to the meta-model, the advice described in Section 7.3.1 differs from the advice usually discussed in the context of AspectJ. In particular, in AspectJ advice applies to points in the execution of the program. With modeling, we have much more freedom than that, and the semantics of the advice we introduce is different, in that advice can apply to any model element which can be identified in our system, i.e. any model element that is a join point.

7.4 Discussion of Contributions

In this section, we look at each of the contributions we claimed in Chapter 1, and justify them with respect to the examples we've discussed above.

7.4.1 Show that a JPM-Enhanced Meta-Model Can Support Crosscutting Structure in a UML Model

Our planner makes the existing crosscutting structure explicit in the model, relying on the additions we have made to the meta-model, as discussed in Chapter 5. In particular, we make use of the "crosscuttingJoinpoints" field of a Joinpoint to record the crosscut-by set for each element.

Our framework also enables the addition of new kinds of crosscutting structure (model elements) with less work than would be required with established modeling tools like RSA. Section 7.3 addresses this exact issue, where we discuss how support for advice and pointcuts can be added. We believe that we could add support for these elements to our meta-model and planner with a couple of weeks worth of work, while there is still no support in RSA for advice, regardless of the increasing

interest in aspect-oriented implementation and design.

From making the changes to the meta-model, as well as the implementation of the planner, we saw that crosscut-by sets are a good way of supporting and exposing crosscutting structure. The implementation was straightforward, as is access to the elements in the set after the planner has made its passes. All the different kinds of crosscutting elements are treated the same, which allows analysis tools to be more generic in their implementation.

7.4.2 Traversing the Model to Collect Crosscutting is Straightforward

Once the JPM was added to the meta-model, the planner tool implementation was fairly straightforward, as discussed in Chapter 6. The staged implementation allowed us to concern ourselves only with specific diagrams during each of the passes. All model elements of interest have signatures which can be compared in order to determine whether elements match or not, and we have devised matching rules that are applied in the case of roles from collaborations, which complicate the matching process.

7.4.3 The Above Help Modeling Tools to Access, Analyze, and Display Crosscutting Relationships of Interest

Sections 7.1 and 7.2 discussed simple automatic tools we have implemented that help with analysis of existing crosscutting structure. Section 7.3 dealt with the issue of adding support for new crosscutting elements, to the meta-model, JPM, and planner.

There is less we can say about the display of crosscutting relationships, at least in terms of graphical display. This ended up not being the main focus of the project, so we did not explore the issues involved in creating a graphical display. Deciding how each kind of crosscutting relationship will be represented graphically

is another big issue, and we thought it may be too general of a question to address in our work.

7.4.4 Planner Tool

The planner we implemented as part of this thesis was a good proof-of-concept tool to show that using a JPM can help support modeling, with respect to implementation of both a planner and analysis tools. Both versions of the planner were straightforward to implement, and allowed us to more thoroughly explore the JPM, in particular how it helps simplify the development of different automated analysis tools.

7.5 Feedback and Future Work

We have had a number of opportunities to present this work to other researchers and members of industry, and have had some useful feedback, as well as a number of ideas for the directions we could take the project. In particular, the problem discussed in Section 7.1, on the use of messages in sequence diagrams to recommend possible transitions in state machines, is a direct result of feedback from Susan McIntyre, who quoted some of her customers as saying that when creating state machine diagrams, help from the tool would be desirable, especially if behaviour already specified in sequence diagrams could be used to help with this.

After implementing the first stages of the planner for class and sequence diagrams, we were curious as to what other kinds of diagrams are considered the most interesting (and used most frequently) by modelers. State machine and collaboration diagrams were among those recommended to us, in particular by Bran Selic, who has a long history with real-time modeling. Another question that was raised is whether there is a distinction between design-time crosscutting (what we are dealing with here) and run-time crosscutting (addressed by AOP approaches). Advice, discussed in Section 7.3 can be used as an example of this, since just where

advice can apply, and what it means, is different in AspectJ and our framework.

Maged Elaasar, a PhD student also working at IBM, brought to our attention the idea that since our approach is so general, the JPM concept could also be applied to other meta-models, for example domain-specific languages or meta-models. A planner tool similar to ours could then be implemented for the new domain, with the same kinds of benefits. Finally, it was discussions with other researchers that convinced us to switch from EMF to UML2 as the basis for our implementation, in particular Bran Selic and Ken Hussey, the main developer behind UML2.

Chapter 8

Conclusion

We propose bottom-up support for crosscutting structure in UML by adding a simple JPM to the UML meta-model. This modified meta-model simplifies implementation of tool support for exposing and analyzing crosscutting structure, as well as addition of new kinds of crosscutting structure. It also makes models of crosscutting structure more declarative.

Using our meta-model, adding new ITD constructs was a fairly straightforward exercise. All the existing UML model elements, as well as the ITD constructs, were integrated smoothly into the JPM. The planner we implemented was able to effectively expose the crosscutting between model elements in various diagrams. This makes us optimistic that we will be able to support other kinds of crosscutting model structure, such as advice, for which the proposed steps were discussed in Chapter 7.

Through design of the JPM, as well as the planner, we have also gained a better understanding of the crosscutting relationships that are possible between different model elements, which we summarised in a table in Chapter 5.

In terms of implementation, we present a self-contained JPM-enhanced UML meta-model, and an easily extensible planner tool. Building on that foundation we present simple automated model analysis tools, which can provide further task-specific helpful information to the modeler. These tools rely on the output from

the planner to perform their analysis. These tools would be most useful for analysis that relies on information from multiple models, or multiple diagrams within a single model.

We propose an advice semantics in which any kind of join point can be advised, including not just method calls, but also transitions, classes, states, and others, as discussed in Section 7.3. We would like to see the existing JPM and planner expanded by adding support for other UML diagrams, as well as providing support for the elements of existing diagrams that we excluded during our initial implementation.

We would also like to see the analysis tool suite extended with tools similar to the two we've already implemented. Following one of the suggestions we discussed in Section 7.5, it would also be interesting to apply the approach we presented to a different meta-model, by following the steps for designing a JPM and implementing a planner, which we described in this thesis.

We believe that adding a JPM to a meta-model is promising both in terms of support for existing kinds of crosscutting, as well as addition of support for new kinds of crosscutting, like aspects and advice. Our proof-of-concept implementation demonstrates this. We have also presented ideas for future work, as well as possible automated tools that would take advantage of the crosscutting structure exposed by the planner.

Bibliography

[1] E.P. Andersen and T. Reenskaug. System design by composing structures of interacting objects. In *Proceedings of ECOOP*, pages 133–152, London, UK, 1992. Springer-Verlag.

[2] AspectJTeam. The aspectj programming guide. http://eclipse.org/aspectj/, 2005.

[3] H. Cannon. Flavors: A non-hierarchical approach to object-oriented programming. Technical report, Symbolics Inc., 1982.

[4] C. Chavez and C. Lucena. A metamodel for aspect-oriented modeling. In *Workshop on Aspect-Oriented Modeling with UML at AOSD2002*, Enschede, The Netherlands, April 2002.

[5] S. Clarke, W. Harrison, H. Ossher, and P. Tarr. Subject-oriented design: Towards improved alignment of requirements, design and code. In *Proceedings of the conference on Object-Oriented Programming: Systems, Languages, and Applications*, pages 325–339. ACM Press, November 1999.

[6] S. Clarke and R.J. Walker. Composition patterns: an approach to designing reusable aspects. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 5–14, Toronto, Ontario, Canada, 2001.

[7] G. Georg, R. France, and I. Ray. Composing aspect models. In *The 4th AOSD Modeling with UML Workshop*, October 2003.

[8] Y. Han, G. Kniesel, and A. Cremers. Towards visual aspectj by a meta model and modeling notation. In *6th International Workshop on Aspect-Oriented Modeling*, 2005.

[9] J. Hannemann and G. Kiczales. Design pattern implementation in java and aspectj. In *Proceedings of OOPSLA '02, ACM SIGPLAN Notices*, 2002.

[10] S. Herrmann. Composable designs with ufa. In *Aspect-Oriented Modeling with UML workshop at the 1st International Conference on Aspect-Oriented Software Development*, Enschede, The Netherlands, April 2002.

[11] E. Hilsdale and J. Hugunin. Advice weaving in aspectj. In *International Conference on Aspect-Oriented Software Development (AOSD)*, pages 26–35. ACM Press, 2004.

[12] W.-M. Ho, J.-M. Jezequel, F. Pennaneac'h, and N. Plouzeau. A toolkit for weaving aspect oriented uml designs. In *Proceedings of the 1st international conference on Aspect-oriented software development*, pages 99–105, Enschede, The Netherlands, 2002. ACM Press.

[13] IBM. Rational software architect (rsa). http://www-128.ibm.com/developerworks/rational/products/rsa/, 2005.

[14] I. Jacobson and P.-W. Ng. *Aspect-Oriented Software Development with Use Cases.* Addison Wesley Professional, 2004.

[15] J. Jezequel, N. Plouzeau, T. Weis, and K. Geihs. From contracts to aspects in uml designs. In *Aspect-Oriented Modeling with UML workshop at AOSD*, 2002.

[16] M.M. Kande, J. Kienzle, and A. Stohmeier. From aop to uml - a bottom-up approach. In *Aspect-Oriented Modeling with UML workshop at the 1st International Conference on Aspect-Oriented Development*, 2002.

[17] M. Katara and T. Mikkonen. Refinements and aspects in uml. In *Aspect-Oriented Modeling with UML Workshop at UML Conference*, 2002.

[18] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W.G. Griswold. An overview of aspectj. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 327–355, Budapest, Hungary, 2001. Springer.

[19] G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *ACM International Conference on Software Engineering*, May 15-21, 2005.

[20] D.-K. Kim, R. France, S. Ghosh, and E. Song. A role-based metamodeling approach to specifying design patterns. In *Proceedings of COMPSAC*, pages 452–457, 2003.

[21] S. Krishnamurthi, K. Fisler, and M. Greenberg. Verifying aspect advice modularly. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 137–146, 2004.

[22] J.M. Lions, D. Simoneau, G. Pitette, and I. Moussa. Extending opentool/uml using metamodeling: An aspect oriented programming case study. In *Workshop on Aspect-Oriented Modeling with UML at the UML Conference*, 2002.

[23] H. Masuhara and G. Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 2–28. Springer, 2003.

[24] OMG. Unified modeling language (uml), version 1.5. http://www.uml.org, 2004.

[25] H. Ossher and P. Tarr. Hyper/j: Multi-dimensional separation of concerns for java. In *International Conference on Software Engineering*, pages 734–737. ACM Press, 2000.

[26] R. Pawlak, L. Duchien, G. Florin, F. Legond-Aubry, L. Seinturier, and L. Martelli. A uml notation for aspect-oriented software design. In *Aspect-Oriented modeling with UML workshop at AOSD*, Enschede, The Netherlands, 2002.

[27] I. Philippow, M. Riebisch, and K. Boellert. The hyper/uml approach for feature based software design. In *The 4th AOSD Modeling with UML Workshop*, San Francisco, CA, 2003.

[28] C. Prehofer. Feature interactions in statechart diagrams or graphical composition of components. In *Workshop on Aspect-Oriented Modeling with UML at the UML Conference*, 2002.

[29] Eclipse Project. Eclipse modeling framework (emf). http://www.eclipse.org/emf/, 2005.

[30] Eclipse Project. Uml 2.0 metamodel implementation (uml2). http://www.eclipse.org/uml2/, 2005.

[31] Eclipse Project. Graphical editing framework (gef). http://www.eclipse.org/gef/, 2006.

[32] D. Reifer. Doubts and hopes for aop. In *Communications of the ACM*, volume 45-3, pages 11–12, 2002.

[33] B. Selic. Using uml for modeling complex real-time systems. In *Languages, Compilers, and Tools for Embedded Systems: ACM SIGPLAN Workshop LCTES*, Montreal, Canada, 1998.

[34] Spring. Spring framework. http://www.springframework.org/docs/reference/aop.html, 2005.

[35] D. Stein, S. Hanenberg, and R. Unland. Designing aspect-oriented crosscutting in uml. In *Workshop on Aspect-Oriented Modeling with UML at AOSD*, Enschede, The Netherlands, 2002.

[36] D. Stein, S. Hanenberg, and R. Unland. Position paper on aspect-oriented modeling: Issues on representing crosscutting features. In *Workshop on Aspect-Oriented Modeling at AOSD*, 2003.

[37] G. Straw, G. George, E. Song, S. Ghosh, R.B. France, and J.M. Bieman. Model composition directives. In *Conference on the Unified Modeling Language*, Lisbon, Portugal, 2004.

[38] J. Suzuki and Y. Yamamoto. Extending uml with aspects: Aspect support in the design phase. In *ECOOP Workshop on AOP*, 1999.

[39] T. Tamai, N. Ubayashi, and R. Ichiyama. An adaptive object model with dynamic role binding. In *Proceedings of the International Conference on Software Engineering (ICSE '05)*, St. Louis, MO, 2005.

[40] P. Tarr, H. Ossher, W. Harrison, and S.M. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *International Conference on Software Engineering (ICSE)*, pages 107–119. IEEE Computer Society Press, 1999.

[41] M. Tkatchenko and G. Kiczales. Uniform support for modeling crosscutting structure. In *The Eighth International Conference on The Unified Modeling Language, UML 2005*, pages 508–522, Montego Bay, Jamaica, October 2005. Springer.