MATA: A Unified Approach for Composing UML Aspect Models Based on Graph Transformation*

Jon Whittle[1], Praveen Jayaraman[2], Ahmed Elkhodary[2], Ana Moreira[3], and João Araújo[3]

[1] Dept. of Computing, Lancaster University, Bailrigg, Lancaster LA1 4YW
[2] Dept. of Information and Software Engineering, George Mason University, Fairfax, VA 22030
[3] Dept. of Informatics, FCT, Universidade Nova de Lisboa, 2829-516, Caparica, Portugal
whittle@comp.lancs.ac.uk, praveenjayaraman@yahoo.com, aelkhoda@gmu.edu, {amm,ja}@di.fct.unl.pt

Abstract. This paper describes MATA (Modeling Aspects Using a Transformation Approach), a UML aspect-oriented modeling (AOM) technique that uses graph transformations to specify and compose aspects. Graph transformations provide a unified approach for aspect modeling in that the methods presented here can be applied to any modeling language with a well-defined metamodel. This paper, however, focuses on UML class diagrams, sequence diagrams and state diagrams. MATA takes a different approach to AOM since there are no explicit joinpoints. Rather, any model element can be a joinpoint, and composition is a special case of model transformation. The graph transformation execution engine, AGG, is used in MATA to execute model compositions, and critical pair analysis is used to automatically detect structural interactions between different aspect models. MATA has been applied to a number of realistic case studies and is supported by a tool built on top of IBM Rational Software Modeler.

1 Introduction

Aspect model composition is the process of combining two models, M_B and M_A, where an aspect model M_A is said to crosscut a base model M_B. As such, aspect model composition is a special case of the more general problem of model fusion. A number of techniques and languages have been developed to specify how M_A crosscuts M_B, and, in particular, how M_A and M_B should be composed.

Broadly speaking, there have been, to date, two approaches for specifying aspect model composition. In the first approach, M_A and M_B are composed by defining matching criteria that identify common elements in M_A and M_B and then applying a generic merge algorithm that equates the common elements. Typically, matching criteria are based on easily identifiable properties of model elements. For example,

* This paper is an extended version of a paper previously published at the 2007 International MODELS conference [1]. There was also a workshop paper on the MATA tool [2]. The main new contributions are the section on code generation and the evaluation and discussion section. The section on aspect interactions is also new.

S. Katz et al. (Eds.): Transactions on AOSD VI, LNCS 5560, pp. 191–237, 2009.

two class diagram models can be merged by equating classes with the same name. Examples of this approach include Theme/UML [3] as well as work by France et al. [4]. In the second approach, mechanisms for specifying and weaving aspects from aspect-oriented programming (AOP) are reused at the modeling level. There has been a significant amount of research, for example, that identifies a joinpoint model for a modeling language and then uses the AspectJ advices of before, after, and around for weaving. Examples of this type include [5, 6].

These two kinds of approaches are not always sufficient. A merge algorithm in the first approach based on general matching criteria will never be expressive enough to handle all model compositions. Matching by name, for example, may not work for state diagrams. Given two states with the same name, the states may need to be merged in one of a variety of ways depending on the application being modeled: (1) the two states represent the same thing, which implies making the states equal; (2) the two states represent orthogonal behaviors of the same object, which implies enclosing the states by a new orthogonal region; (3) one state is really a submodel of the other, which implies making one state a substate of the other; and (4) the behaviors of the two states must be interleaved in a complex way, which implies weaving the actions and transitions in a very application-specific way to achieve the desired result. Only the first of these can be accomplished based on merge-by-name. Furthermore, these are only four of the many possible options, and so it is not generally sufficient to provide a number of pre-defined merge strategies. In practice, to overcome this problem, the modeler may additionally specify what Reddy et al. [7] call composition directives—that is, operators that override the default merge algorithm. However, understanding the interactions between the default algorithm and the composition directives is a difficult task, and, in particular, does not work easily for behavioral models (cf. [8]).

In the second approach, specific elements in a model are allowed to be defined as joinpoints and others are not. For example, in state diagrams, some approaches [5] define actions as joinpoints. Others, however, define states as joinpoints [9]. One could even imagine more complex joinpoints, such as the pointcut of all orthogonal regions. (This pointcut might be used, for example, by an aspect that sequentializes parallel behaviors.) Defining only a subset of a model's elements as joinpoints seems to be overly restrictive. In addition, limiting advices to before, after, and around (as is done, for example, by both [5] and [9]) is also rather restrictive since it may be desired to weave behavior in parallel or as a sub-behavior of a behavior in the base.

This paper takes a step back to reassess the requirements for aspect modeling languages. The result is the technique and tool MATA (Modeling Aspects Using a Transformation Approach), which tackles the above limitations by viewing aspect composition as a special case of model transformation. In MATA, composition of a base and aspect model is specified by a graph rule. Given a base model, M_B, crosscut by an aspect model, M_A, a MATA composition rule merges M_A and M_B to produce a composed model M_{AB}. The graph rule r: LHS $\rightarrow$ RHS defines a pattern on the left-hand side (LHS). This pattern captures the set of joinpoints, i.e. the points in M_B where new model elements should be added. The right-hand side (RHS) defines the new elements to be added and specifies how they should be added to M_B. MATA graph rules are defined over the concrete syntax of the modeling language. This is in contrast to almost all known approaches to model transformation, which typically

define transformations at the meta-level, that is, over the abstract syntax of the modeling language. The restriction to concrete syntax is important for aspect modeling because a modeler is unlikely to have enough detailed knowledge of the UML metamodel to specify transformations over abstract syntax.

MATA currently supports composition for UML class, sequence, and state diagrams. In principle, however, it is easy to extend MATA to other UML models (or, indeed, other modeling languages as long as a metamodel for the language exists) because the idea of using graph rules is broadly applicable. MATA makes no decisions on joinpoint models, for example, which would limit the approach to specific diagram types.

One advantage of using graph transformations for aspect model composition is that graph transformations are a well-understood, formal technique with formal analysis tools available. In particular, critical pair analysis can be used to automatically detect dependencies and conflicts between graph rules. MATA applies critical pair analysis to detect interactions between aspects. This can be done because each aspect is represented as a graph rule and so the problem of aspect interaction can be stated in terms of dependencies between graph rules. Not all kinds of interactions can be detected–the technique is limited to structural rather than semantic interactions–but critical pair analysis offers a fully automatic, lightweight method for finding these structural interactions between aspect models.

This paper gives a full description of the MATA language for aspect model composition, its underlying graph transformation representation, and the use of critical pair analysis for detecting aspect interactions. It also describes the tool support for MATA, which is implemented on top of IBM Rational Software Modeler. The contributions of this paper can be divided into three categories as follows:

1. A unified, expressive approach for aspect model composition:
   - MATA is agnostic with respect to the modeling language to be composed as long as there is a well-defined metamodel for this language.
   - MATA is more expressive than previous approaches because it views aspect model composition as simply a special case of model transformation.
   - MATA handles both structural and behavioral models in the same way.
2. A usable graph transformation language for aspect model composition:
   - Graph rules in MATA are written in the concrete syntax of the modeling language, not in the abstract syntax. This allows them to be specified graphically in a way that is very similar to defining models for the underlying modeling language.
   - Graph rules in MATA provide support for *sequence pointcuts*, where a pointcut is a *sequence* of elements, which allows rich specification methods available in graph transformations to be available for aspect model composition, but in a way that is accessible to model developers.
3. An automatic technique for detecting structural interactions between aspect models:
   - Critical pair analysis has been applied to detect interactions between models given as UML class diagrams, sequence diagrams, and state diagrams.

The paper is organized as follows. Section 2 motivates why a new, unified, and expressive approach to aspect model composition is needed. Section 3 provides background on graph transformations necessary to describe the MATA approach. Section 4 describes the MATA language and Sect. 5 explains the application of critical pair analysis for detecting aspect interactions. Section 6 presents an extended example illustrating MATA and is followed, in Sect. 7, by a description of MATA tool support and, in Sect. 8, by a discussion of how MATA has been applied in practice. Conclusions follow in Sect. 9.

2 Motivation

This section motivates why existing approaches to aspect model composition are not expressive enough. The goal here is to show either that existing approaches cannot specify compositions in certain cases or that they cannot do it in an intuitive way. To illustrate this, we use a simple but non-trivial, example of an aspect model composition and argue that previous approaches are non-optimal.

Note that this paper takes a rather general definition of the term aspect such that any view of the system can be called an aspect. This means that many existing decomposition techniques (e.g. use cases and features) can be seen as aspects. This interpretation is consistent with that of many authors [6, 10, 11]. The examples in the paper will reflect this definition. This general view in particular means that our technique for handling aspect models works just as well for crosscutting and non-crosscutting concerns. In other words, we handle aspectual and non-aspectual concerns in a uniform way.

Figure 1 is an example of using UML use cases to maintain separation of concerns in a distributed application. The idea here (following [6]) is that the use case models are maintained separately throughout system development and that they can be composed at any time using aspect composition. The LHS is a use case for calling a remote service and consists of a state dependent class ServiceController and a state diagram that defines its behavior. The RHS is a use case for handling a network failure, which contains the same class ServiceController, but with a different set of attributes and a different state diagram. This second use case describes a limited number of attempts to retry a service.

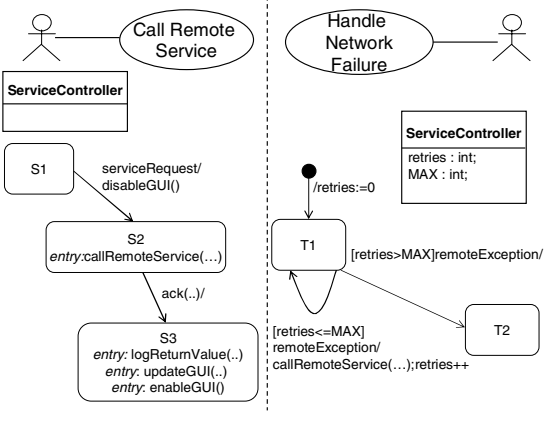

Fig. 1. Maintaining Use Case Separation of UML Models

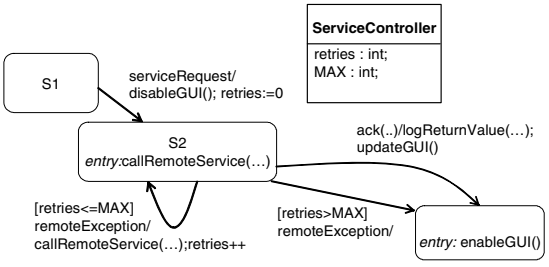

Fig. 2. Desired Composition of State Diagrams from Figure 1

The RHS crosscuts the LHS in the sense that whenever *callRemoteService* appears on the LHS, the RHS behavior should be used to handle a failure. This turns out to be a non-trivial example of crosscutting behavior. Prior to calling the remote service on the LHS, a GUI is disabled (via the action *disableGUI*). The GUI is only re-enabled (via the action *enableGUI*) once the remote service has been called **successfully**—the service call succeeds, a log is taken, and the GUI is updated before the GUI is re-enabled.

Now, consider the desired result from composing the RHS with the LHS—this is shown in Fig. 2. Note that when failure-handling is incorporated, what is now needed is that the GUI should be re-enabled whether the calling of the remote service succeeds or not. That is, even if the maximum number of retries is exceeded, *enableGUI* must still occur. Furthermore, logging and updating must only occur if the service call succeeds.

Capturing this composition is quite difficult if a composition model based on that of AspectJ is used. Existing work on (AOM) might, for example, define a joinpoint as the occurrence of the action *callRemoteService*. One might then insert behavior after or around this joinpoint in such a way that *enableGUI* is called whether or not the service call succeeds, and that logging/updating is not called in the failure scenario. This is possible but would really require the definition of two separate aspects, each with separate joinpoints—one joinpoint being the state containing action *callRemoteService()* to which the *[retries<=MAX]* transition would be added, and the other being the transition with event *ack/* to which *logReturnValue()* and *updateGUI()* would be added as actions. In addition, one would need to use an around advice to ignore the first two entry actions in state *S3*. The effect is that the failure handling model on the RHS gets broken into pieces, thus becoming harder to understand the failure handling aspect in its own right. This effect goes against many of the ideas of modeling in that models ought to be easily readable.

Capturing this composition using some kind of default merge algorithm is also difficult. For example, one could proceed by defining a correspondence between states and then merging those states. The obvious thing to try would be to equate *T1* and *S2*, but the merge based on this correspondence would fail to re-enable the GUI if the maximum number of retries is reached. If one tries to solve this, in addition, by equating *T2* and *S3*, then the GUI will be re-enabled, but the logging and updating will occur even if the remote service call fails, which is contrary to the requirements given

above. Therefore, composition directives would be needed to refactor the result of equating states. The problem with such composition directives is that it is hard to know exactly which directives to use because one has to first visualize the result of the merging. For large state diagrams, it becomes very complex to be able to predict where composition directives will need to be applied after the merge is complete.

As it turns out, one neat way to handle this example is by defining a so-called *sequence pointcut* [12]. A sequence pointcut should be used when it is not enough to consider a single element as a joinpoint, but instead, the joinpoint should be a *sequence* of elements. In this example, the key sequence starts with disabling the GUI and ends with re-enabling the GUI. This is because the GUI must be both disabled and re-enabled whatever the outcome of the remote service call. If one could specify that the pointcut is the sequence of actions/events between *disableGUI* and *enableGUI*, then one can easily capture the fact that the aspect should only apply to sequences where the GUI is first disabled and then later re-enabled. This allows one to specify, *on the same diagram*, that the failure handling (i.e. the aspect) behavior begins after *disableGUI* and ends with *enableGUI*. Further details on how sequence pointcuts can be defined in MATA are given in Sect. 4. Sequence pointcuts are not currently possible with most AOM approaches[1], although some AOP languages do support them [12].

More generally, when composing crosscutting state diagrams, it may be desirable to use advices that are more expressive than before, after or around. For example, an aspect state diagram may need to be composed *in parallel* with a base state diagram, or an aspect state diagram may need to be inserted *inside* a state in the base diagram (i.e. the base state becomes a composite state). In fact, composition should allow two diagrams to be composed using any of the syntactic constructs of the modeling language. In the case of state diagrams, for example, composition could be achieved using orthogonal regions, composite states, or even history states.

In other words, aspect-oriented model composition may require models to be composed in complex ways rather than just before or after each other. Previous approaches to AOM do not support such complex compositions. It is for this reason that we propose a new model composition language in this paper.

3 Background

Before going on to explain the details of the MATA language, this section first presents necessary background material. MATA is based on the technique of graph transformations and so a brief introduction to graph transformations is given in this section. We also briefly describe critical pair analysis, which will be used to detect interactions between aspects.

3.1 Graph Transformations

A graph consists of a set of nodes and a set of edges. A graph transformation is a graph rule $r: L \rightarrow R$ from a LHS graph L to a RHS graph R. The process of applying r

[1] The only known approach that does allow this is joinpoint designation diagrams (JPDDs) [13] but JPDDs do not support expressive advices.

to a graph *G* involves finding a graph monomorphism *h* from *L* to *G* and replacing *h(L)* in *G* with *h(R)*. Graph transformations may also be defined over attributed typed graphs. A typed graph is a graph in which each node and edge belongs to a type. Types are defined in a type graph. An attributed graph is a graph in which each node and edge may be labeled with attributes where each label is a (value, type) pair giving the value of the attribute and its type. In a graph rule, variables may be used to capture a set of possible values and/or a set of possible types.

Graph rules have previously been used for transforming UML models (e.g. UML refactorings [14]). Such work requires that UML models be represented as graphs. The usual approach is to define node types as the metaclasses in the UML metamodel. Graph rules can then be shown graphically using object diagrams.

As an example, Fig. 3 shows a (simplified) fragment of the UML state machine metamodel. A state machine contains 1 or more (orthogonal) regions, each of which contains states. Each transition is from a source to a target state and has a trigger and actions. States may also have actions. A state may contain 0 or more regions. A state is composite if it contains 1 or more regions. If it contains 2 or more regions, then the regions in this state are orthogonal. The *State* metaclass has an attribute *isComposite* indicating whether or not the state is composite. Finally, states, triggers, and actions have names (as represented by a generalization relationship to *namedElement*).

Figure 4 is an example graph transformation which moves all outgoing transitions from a composite state to its substates. The notation used to define this graph transformation is that of [14]. (We defer to [14] for the subtleties of this notation.) Nodes in the graph are given as rectangles. Nodes are attributed and typed and so the UML object diagram notation can be used to represent them. There are two additional notations. First, a set of nodes of a certain type is shown by a stacked rectangle. For example, *regions* is a set of Regions associated with a composite state. Secondly, the cross in the figure is a negative application condition and says that any match against the LHS graph cannot have a substate with a transition trigger called *triggerName*.

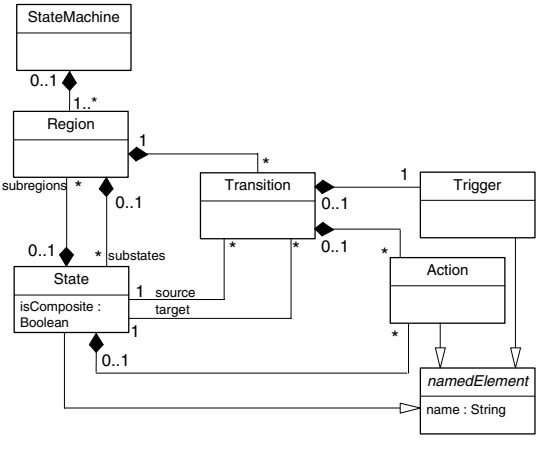

Fig. 3. UML State Machine Metamodel

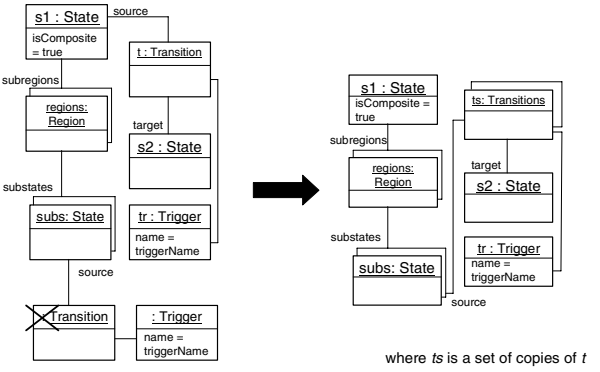

Fig. 4. Graph Rule to Move Down Transitions

The LHS in Fig. 4 matches any graph with at least one composite state with an outgoing transition. Furthermore, there should not be a transition on any of the substates with the same trigger. The RHS redirects the matched transition to all substates (by creating copies) thus moving the transition down in the state hierarchy.

3.2 Critical Pair Analysis

Critical pair analysis is a technique invented for term rewriting systems to check whether a set of rewrite rules is confluent. A set of rewrite rules is confluent if for all x,u,w with $x \gg u$ and $x \gg w$, there exists a z such that $u \gg z$ and $w \gg z$. Here, $\gg$ denotes the application of zero or more rewrite steps–i.e. $x \gg u$ means x rewrites to u in any number of rewrite steps. If a set of rewrite rules is finitely terminating, that is, there are no infinite rewriting sequences, then, confluence implies that all terms have unique normal forms. This in turn implies that, for a given term, the set of rules can be applied in any order and the result will be the same. This is an important property because it allows rules to be applied exhaustively without any concern about interactions or dependencies between rules.

As a simple example, consider a rewrite system consisting of two rules, p_1 and p_2 with p_1: $f(X,X) \rightarrow X$ and p_2: $g(f(X,Y),X) \rightarrow h(X)$, where X and Y are variables. This is not a confluent rewrite system. This can easily be shown by choosing the term $g(f(a,a),a)$ for a constant a, which rewrites under p_1 to $g(a,a)$ and under p_2 to $h(a)$. Since there is now no way to rewrite $g(a,a)$ and $h(a)$ to the same term, the rule set is not confluent.

Critical pair analysis examines potential overlaps between rules. For instance, if X is unified with Y, p_1 and p_2 overlap at $f(X,X)$. This leads to two possible rewriting results for the term $g(f(a,a),a)$ because either of the two rules can be applied. $(g(a,a),h(a))$ is called a critical pair and corresponds to the two possible ways of rewriting $g(f(a,a),a)$. By analyzing all possible critical pairs, all potential overlaps are examined, i.e. all ways that might lead to divergent results are analyzed. In essence, therefore, critical pair analysis is a way of detecting structural interactions between rules.

Formally, critical pairs can be defined as follows. If $x \rightarrow y$ and $u \rightarrow v$ are two rewrite rules with no variables in common (rename them if there are), and if x_1 is a non-variable subterm of x unifiable with u via most general unifier θ, then the pair $y\theta$ and the result of replacing $x_1\theta$ in $x\theta$ by $v\theta$ is called a critical pair.

Critical pair analysis has been adapted to graph rules—see, for example, [15]. In the context of MATA, since an aspect is a graph rule, critical pair analysis can be applied to detect overlaps, i.e. interactions, between aspects. When applied exhaustively, critical pair analysis will find all aspects (i.e. graph rules) that are in conflict or are dependent, where conflict and dependency are defined as follows:

- Aspect A conflicts with aspect B if the application of aspect A prevents aspect B from being applied.
- Aspect B is dependent on aspect A if the application of aspect A is necessary for aspect B to be applied.

Examples of conflicts and dependencies for UML aspect models are given in Sect. 5.

4 Specifying and Composing Aspect Models with MATA

This section describes how to specify and compose aspect models with MATA. MATA considers aspect composition as a special case of graph transformation. The key difference with existing graph transformation approaches such as FUJABA [16] and VIATRA2 [17] is that these approaches define transformations using the abstract syntax of the modeling language. For example, the transformation in Fig. 4 refers to metaclasses such as *Region* and *State*. Even for a simple transformation such as the one in Fig. 4, the use of abstract syntax soon becomes complicated and it becomes very difficult to specify such rules correctly. This is particularly true for UML sequence diagrams because the metamodel for interactions in UML is quite complicated. Since MATA is targeted toward model developers, not metamodeling experts, its aspect models must be specified in a way that is intuitive for users unfamiliar with the intricacies of the UML metamodel. This means that aspect rules should be specified using the concrete syntax of UML rather than UML metaclasses.

For the most part, specifying a graph rule over UML using concrete syntax is straightforward. As long as a metaclass has a concrete visualization, users can draw diagrams using this visualization and it can be translated automatically to the relevant metaclass. Abstract metaclasses, which do not have a concrete syntax realization, cannot be drawn using concrete syntax. Such abstract metaclasses cannot be used in MATA and so MATA should not be viewed as a general purpose transformation language, but rather a transformation language specialized toward aspect model composition. For aspect model composition, abstract metaclasses do not need to be used.

MATA aspect models, therefore, are graph rules written in concrete syntax that are translated into equivalent abstract syntax for the purposes of executing the transformation. MATA does include some extensions to UML's concrete syntax that are necessary to support its notion of sequence pointcuts. Recall from Sect. 1 that sequence pointcuts are used to match against a sequence of model elements in the base. In MATA, this can be a sequence of transitions in state diagrams, or it can be a sequence of messages in sequence diagrams. Sequence pointcuts turn out to be a very powerful

mechanism for specifying aspects in a way where the aspect is as ignorant as possible of elements in the base.

The remainder of this section explains the MATA language in detail. First, an overview of how to specify aspects in MATA is presented. This is followed by details on specifying joinpoints and advices in MATA.

4.1 An Overview of Using Aspects in MATA

Figure 5 provides an overview of how aspect models are specified in MATA. A *model slice* is defined as a collection of structural and behavioral models (UML class diagrams, state diagrams, and sequence diagrams) that capture a particular view of the system. The base model slice captures the core system model with crosscutting concerns removed. An aspect model slice captures the models for a particular crosscutting concern.

The base model slice is composed of a set of base models. Similarly, an aspect model slice is composed of a set of aspect models. Base models are written in standard UML. Aspect models are written in the MATA language and are defined as increments of the base models or other aspect models. Each aspect model describes the set of model elements affected by the aspect (i.e. the joinpoints) and how the base model elements are affected (i.e. the advices). Note that an aspect model can only be defined as an increment of a model of the same type; for example, sequence diagram aspects can extend base sequence diagrams but not base state diagrams.

The following process can be used to develop and compose aspect models. The modeler first develops the base model slice and a set of aspect model slices. Each aspect model slice is written as an increment over the base model slice or as an increment over other aspect model slices. The user then invokes the MATA composition engine to compose the base slice with a selected subset of the aspect slices. Before performing the composition, MATA applies critical pair analysis to detect interactions within the set of chosen aspect slices. Interactions can be detected

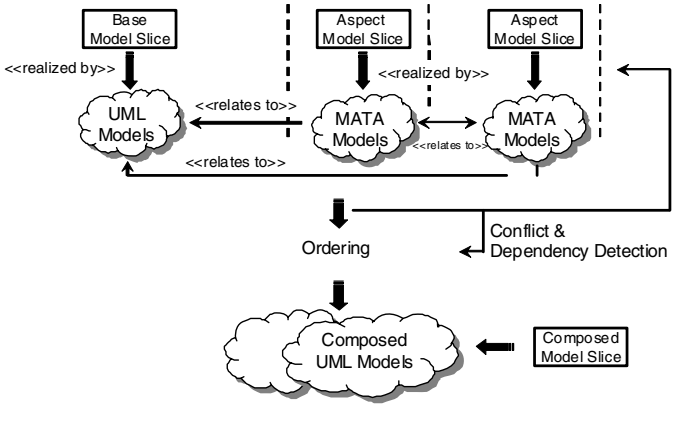

Fig. 5. An Overview of MATA

between models of the same type. The results of this analysis are provided to the user and result in one of the following three conclusions:

1. There are no interactions.
2. There are interactions that mean that the aspects must be applied in a particular order. The user then specifies this order.
3. There are interactions that cannot be resolved by applying the aspects in a particular order. Instead, either the base or aspect models must be modified to remove these unwanted interactions.

Once all interactions have been resolved either by (2) or (3), the modeler instructs MATA to compose the chosen aspects with the base. The result is a new model slice that can be inspected, analyzed, or from which code can be generated. Note that there is no necessity to actually compose the models. The key point is that the MATA specification contains a precise description of the aspect and base relationships. This description can either be used in composition or can be used to generate aspect-oriented code by generating the code for each model slice and generating the AOP code that specifies how to weave the aspect code into the base. In fact, MATA comes with a code generator that does exactly this, resulting in AspectWerkz [18] code (see Sect. 7).

Note that MATA does not address how to partition a problem into an appropriate set of aspect slices, i.e. how to decide on the right set of aspects. This is a more general problem, which is out of the scope of this paper, but existing techniques for identifying aspects during requirements engineering, such as [19], could be applied to identify requirements-level aspects and then model these aspects during the analysis and design phases using MATA.

4.2 Joinpoints, Advices and Aspects in MATA

There are no explicit joinpoints in MATA. Any model element can be a joinpoint and pointcuts are defined as patterns over these model elements. Similarly, there are no restrictions on the advices in MATA. In particular, MATA is not limited to before, after, and around advices. Instead, any model element of the underlying model language can be used. For example, composition in parallel is allowed in state diagrams using orthogonal regions.

Hence, an aspect model in MATA consists of two parts, a *pattern* and a *composition specification*. Application of an aspect model to an existing base model is done in two stages:

1. Find a match for the pattern in the base model.
2. Modify the base model at the matched locations according to the composition specification.

This is just a standard application of graph transformation techniques.

4.2.1 MATA's Pattern Language

A pattern in MATA can be either a simple pattern or a complex pattern. (This distinction is made purely for presentational purposes.) A simple pattern is just a UML model with some elements marked as *pattern variables*. Pattern variables are typed over UML metaclasses and are regular expressions prefixed with a vertical bar "|" to

denote that they are variables. For a simple pattern, matching the pattern against a UML base model consists of finding an instantiation of the pattern variables in the aspect model such that the structure of the aspect model is preserved. Standard efficient algorithms for matching in graph transformations can be used for this [20].

Complex patterns include patterns that define sequence pointcuts. Sequence pointcuts are currently provided for state diagrams and sequence diagrams, and are described next.

Sequence pointcuts in state diagrams

Sequence pointcuts in state diagrams are a general way of matching against multiple elements at once. This is particularly useful, for example, when one wants to match against a sequence of transitions beginning and ending with a particular event, but where the events on intermediate transitions are unimportant. Sequence pointcuts introduce new concrete syntax into patterns because multiple model elements must be matched against. However, the concrete syntax is extended in as minimal a way as possible.

A state diagram sequence pointcut, therefore, is an abstract representation of a family of state diagrams and contains pattern variables. In complex patterns representing sequence pointcuts, pattern variables have multiplicities. A pattern variable |X has a multiplicity of one. A pattern variable $|X^+$ has a multiplicity of one or more. A complex state diagram pattern matches a state diagram if all the pattern variables can be instantiated to elements of the state diagram in a way that preserves the variable's metaclass and multiplicity.

State Diagram Sequence Pointcut Syntax. We denote the type of a pattern variable by (|X : T). Only the metaclasses in the list below are allowed to have pattern variable multiplicities. We assume the metamodel of Fig. 3 in the remainder of this paper.

1. (|X : State) matches against a single state. ($|X^+$: State) matches against one or more states and also matches the transitions between these states. More precisely, $|X^+$ will match a fully connected substate machine–that is, each state included in the match must be connected by at least one transition to another state included in the match.
2. (|X : StateMachine) matches a single state machine. ($|X^+$: StateMachine) is not allowed (because it is unnecessary).
3. (|X : Action) matches a single action. ($|X^+$: Action) matches a sequence of one or more actions.
4. (|X : Trigger) matches a single event. ($|X^+$: Trigger) matches a sequence of one or more events.
5. (|X : Region) matches a single orthogonal region. ($|X^+$: Region) matches one or more regions within the same composite state.

Whenever possible, the concrete syntax of a pattern variable is the same as the UML concrete syntax of its type. See Fig. 6 for examples.

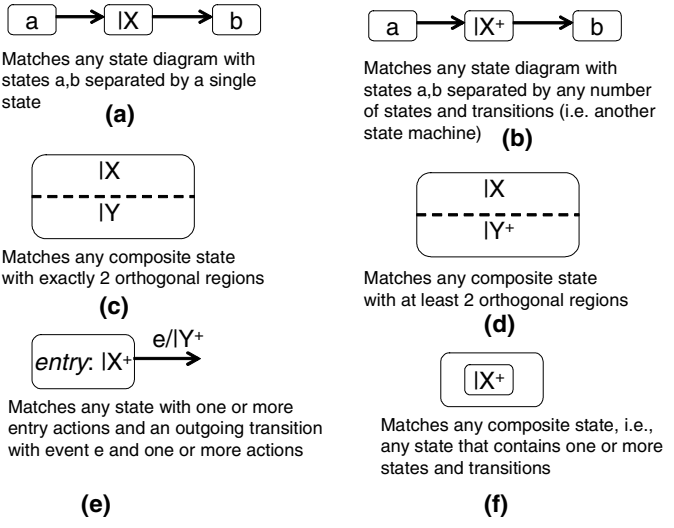

Fig. 6. State Diagram Pattern Examples

Figure 6(a), for example, matches any sequence of states starting with a state named *a*, ending with a state named *b*, and with another state in between (different from *a* and *b*). In contrast, the variable $|X^+$ in Fig. 6(b) matches one or more states in between *a* and *b* as well as any transitions between those states. This means that $|X^+$ represents any number of states and transitions with at least one of those states connected to the incoming transition shown, and at least one state connected to the outgoing state shown. In a similar way, Fig. 6(c) and 6(d) show how to match against a specific number of regions and one or more regions, respectively. Figure 6(e) is self-explanatory. Figure 6(f) matches a state which contains a state machine, i.e. there must be at least one substate, but the composite state may contain any number of substates and transitions.

Note that, for any simple pattern, the name of the pattern variable may be omitted–so, Fig. 6(a) would be equivalent if |X was removed.

State Diagram Sequence Pointcut Semantics. The pattern-matching semantics for state diagram sequence pointcuts is given by mapping each pattern to a typed graph consisting of instances of the appropriate metaclasses. If a pattern element has a multiplicity of one, it maps to a single instance of its metaclass. If it has multiplicity of one or more, it maps to a set of instances. To illustrate, Fig. 7 shows the mapping to metaclass instances for the patterns given in Fig. 6(c) and (d). The first pattern will match any composite state with exactly two orthogonal regions. The second pattern will match any state with at least 2 regions.

A slight complication is introduced by the use of $|X^+$ to match against a set of states and transitions in Fig. 6(b) and (f). In Fig. 6(f), for example, instead of mapping $|X^+$ to a set of instances of State, it must be mapped to an instance of Region containing any

number of instances of State and Transition. This issue arises because of the peculiarities of the UML metamodel.

State Diagram Pattern Example. Figure 8 shows the state diagram pattern required in the example of Sect. 2. Recall that a sequence pointcut was deemed to be useful. The figure illustrates how to specify a sequence from *callRemoteService* to *enableGUI*. The pattern variable $|X^+$ matches against any number of actions in the target state of the transition but will not match against *enableGUI()*. The effect is that the state diagram pattern matches any sequence starting with the *callRemoteService()* action, followed by a transition, and by one or more entry actions, and ending with the action *enableGUI()*.

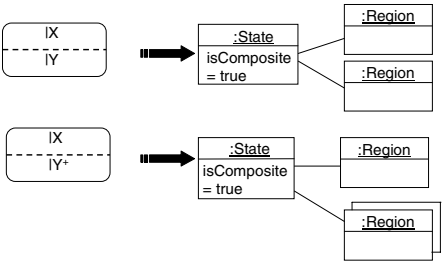

Fig. 7. Metaclass Instance Representation of Patterns

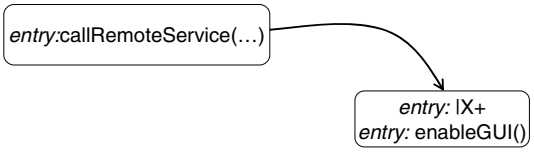

Fig. 8. State Diagram Sequence Pointcut for Figure 1

Sequence pointcuts in sequence diagrams

Sequence pointcuts are also supported in sequence diagrams, but are somewhat simpler. A sequence pointcut here corresponds to any sequence of ordered model elements, including messages and combined fragments. To match the concrete syntax closely, a new interaction fragment is introduced, with interaction operator **any**. An **any** fragment is a variable that will match against any sequence of messages and/or combined fragments. In Fig. 9, the *Call Remote Service* use case from the LHS of Fig. 1 is instead modeled as a sequence diagram. This is shown on the top half of Fig. 9. The bottom half of Fig. 9 gives a sequence pointcut equivalent to that shown in Fig. 8, but for sequence diagrams. Note how this sequence pointcut is agnostic about the messages occurring in between *callRemoteService* and *enableGUI*.

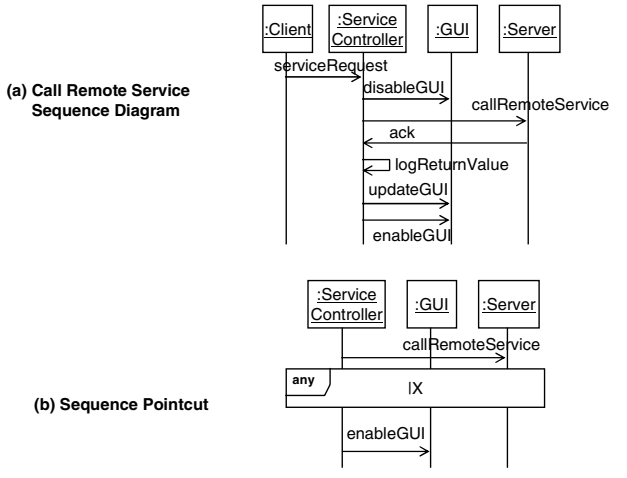

Fig. 9. Sequence pointcuts for sequence diagrams

4.2.2 MATA's Composition Specification Language

MATA's pattern language identifies model elements in the base models that are crosscut by the aspect models. MATA also gives a way to define how model elements from the aspect should be composed with model elements from the base. MATA represents an aspect as a graph rule $r: L \rightarrow R$, where L and R are UML models, as a single UML model in which model elements may be annotated with one of three stereotypes— <<create>>, <<delete>> or <<context>>.

Given a pointcut definition as a MATA pattern, model elements from the aspect that should be added to the pattern are marked with the <<create>> stereotype. Similarly, elements may be removed using the <<delete>> stereotype. Simple examples are shown in Fig. 10 for state diagrams. In (a), the pointcut is any state (where an explicit pattern variable !X has been omitted) and the aspect elements added are a state *a* and a transition to *a*. In (b), the pointcut is any pair of states with a transition between them, and the aspect element is a superstate that is added so that it contains these (and only these) two states. In general, <<create>> and <<delete>> can be used to add (or remove) any kind of aspectual model element. For example, an aspect could be added as an orthogonal region to an existing base model that matches a state pattern—see Fig. 10(c).

The use of <<create>> is "optimized" in the sense that if a state is stereotyped as <<create>>, then any of its substates or transitions are also created. Hence, in Fig. 10(a), the transition is created but does not need to explicitly be given a <<create>> stereotype. This optimization reduces the number of stereotypes a user must specify. However, in Figs. 10(b) and 10(c), the user wants to wrap a composite state around existing states. To stop these substates from being created, they are stereotyped as <<context>>. <<context>> therefore overrides the "optimization". In particular, in Fig. 10(b), although the outer state is marked with <<create>>, the use of <<context>> means that the two inner states are matched against rather than created.

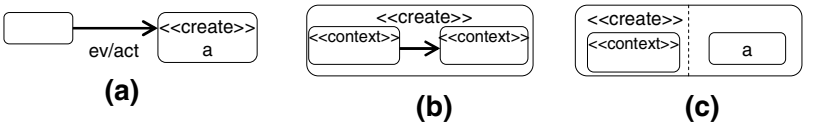

Fig. 10. Examples of Composition Specifications

MATA's composition stereotypes can also be applied to class diagrams and sequence diagrams. We illustrate with some examples of sequence diagram composition specifications.

Figure 11 gives an example MATA aspect rule to add parallel behavior in a sequence diagram. Figure 11(a) is the MATA rule itself and (c) shows the application of the rule to a particular example. (In (a), the lifelines are pattern variables—as before, the pattern variables do not need to be explicitly named.) Figure 11(a) has two parts to it—the pattern to match against and elements to add. As with state diagrams, <<create>> in MATA sequence diagrams is "optimized" so that if <<create>> is applied to a combined fragment, it will also be applied to everything inside the fragment unless it is marked with <<context>>. Similarly, if <<create>> is applied to a lifeline, it is also applied to any messages that are sent to or are received by this lifeline. <<delete>> works in the same way.

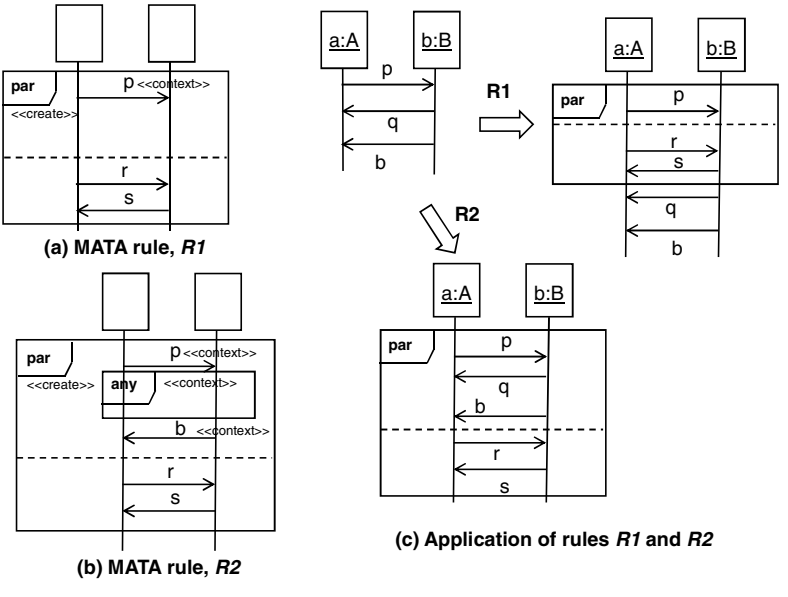

Fig. 11. MATA Rules

Hence, for the **par** fragment in Fig. 11(a), <<create>> also applies to messages *r* and *s*. To avoid <<create>> being applied to *p*, it is marked with <<context>>. Therefore, the match defined in Fig. 11(a) is any pair of lifelines with a message *p* from one lifeline to the other. The effect of applying the rule in Fig. 11(a) is to introduce a new **par** fragment around all instances of message *p*, and this new fragment will have messages *r* and *s* occur in parallel with *p*. This is shown in Fig. 11(c).

Figure 11 also shows an example of how sequence pointcuts and composition specifications can be used together in MATA. The rule R2 in Fig. 11(b) will match any two lifelines with messages *p* and *b* with any number of messages between *p* and *b*. (Note that the messages matched by the **any** fragment need not have the same sender and receiver lifelines as *p* and *q*–that is, the lifelines across which **any** is drawn are irrelevant.) The result of applying the rule is shown in Fig. 11(c). Note how the result is different than if rule R1 is applied. For R2, the pointcut is the sequence of messages *p, q, b*, and so these messages all appear in the first operand of the **par** fragment.

Semantics of MATA's composition language

As already indicated, the use of the <<create>> and <<delete>> stereotypes are "optimized" to reduce the burden on the modeler of applying these stereotypes. This "optimization" is governed by the rules of neighborhood; for example, if <<create>> is applied to a model element, it is also applied to all of its neighbors. Similarly, it holds true for <<delete>> and <<context>>. Since this optimization process can get quite involved for complex examples, we define here precisely how the optimization works.

The semantics is defined by transforming an aspect into the equivalent graph rule in the form LHS → RHS. This is done in two steps. First, the stereotypes <<create>>, <<delete>> and <<context>> are propagated throughout the aspect model. Second, the stereotypes are eliminated by transforming the aspect into a graph rule.

In the first step, each stereotype is propagated to its neighbors. A neighbor may be an immediate neighbor or a remote neighbor. For a given model element, its immediate neighbors are all those related model elements that are considered strongly related to it. For example, the trigger events on a transition are strongly related to the transition itself because they cannot exist without the transition. States are strongly related to their transitions because if a state is deleted, then its transitions must be deleted lest a hanging transition remains. Container states are considered to be strongly related to the elements they contain. For example, composite states are strongly related to the contained states. On the other hand, a transition is not strongly related to its target or source state because the transition can be deleted without deleting the states and the result will still be a well-formed model.

Table 1 gives the immediate neighbors for the model elements considered in this paper.

A remote neighbor of a model element is any neighbor of an immediate neighbor of the model element. The immediate neighbors are designed both to ensure termination of the propagation process and, as much as possible, to avoid aspects introducing ill-formed models.

There are two precedence rules that must be taken into account during the propagation process. This is because a model element may end up with more than one MATA stereotype either because different stereotypes were propagated from different directions or because the user has specifically assigned a stereotype. In the former case, <<context>> always takes precedence over <<delete>> or <<create>> and so <<delete>> and <<create>> are removed in this case. In the latter case, the user-defined stereotype always takes precedence. For example, if a model element is marked as <<delete>> by the user but <<context>> is propagated to it, then <<context>> is removed. If the propagation process ends up with <<create>> *and* <<delete>> both applied to the same element, then there is an inherent inconsistency in the aspect rule and the rule should not be applied. This can happen, for instance, if the user specifies that a state should be deleted but an incoming transition to that state should be created. Obviously, one cannot create a new transition to a state that is marked for deletion.

The following summarizes the propagation process.

for each MATA-stereotyped model element, m, in the aspect model:
    let N be the set of immediate neighbors of m;
    propagate the MATA stereotypes of m to all elements of N;
    for each n in N,
        apply the propagation process
    end foreach
end foreach

for each model element, m, in the aspect model:
    eliminate MATA stereotypes according to the precedence rules
    if m is stereotyped with both <<create>> and <<delete>>, STOP
end foreach

The second step of the semantics definition is to construct the equivalent graph rule. This is done easily. <<create>> and <<delete>> are simply a way of representing both the LHS and RHS of a graph rule on the same diagram. The familiar LHS→ RHS notation can be obtained by considering the LHS as all elements either with no stereotype or with <<context>> or <<delete>>. The RHS is the LHS but with the <<create>> elements added and the <<delete>> elements removed. The Appendix discusses how we do the conversion from UML models in concrete syntax to typed graphs.

Although the propagation algorithm is designed as much as possible to ensure the result of applying an aspect is a well-formed model, there are still situations where this cannot be guaranteed. For example, if state *X* has a transition to *Y* and both *Y* and the transition are marked as <<context>>, whereas *X* is marked as <<create>>, then this rule looks for an existing transition with some undefined source state and creates a new source state for the transition. However, a transition cannot have two source states. We leave as future work to define constraints over how rules are defined that would either avoid such rules or alert the user. Experience has shown that such rules rarely occur in practice.

Table 1. Immediate Neighbors for Some Common Model Elements

The table should be read as follows. The second column lists model elements. For each of these elements, if a MATA stereotype is applied to it, then all elements from the third column are also given the stereotype. So, for example, if a class has a <<delete>> stereotype, all associations connected to this class will also be deleted.

Diagram	Model element	Immediate Neighbors
Class diagram	Class	Connected Association, Contained Attribute, Contained Operation
	Aggregation or Composition Association	Aggregate or Composite Classes
	Generalization	Child Classes
	Other Association	None
State diagram	State	Incoming or Outgoing transition, Substates, Entry or Exit Actions
	Transition	Event on the transition, Action on the transition, Guard on the transition
	Event	None
	Action	None
Sequence diagram	Combined fragment	Model elements contained in the fragment
	Lifeline	Incoming or outgoing message
	Message	None

4.3 MATA Example

Finally, in this section, we return to the remote service call example introduced in Fig. 1. We now consider how to specify this aspect composition in MATA. The base model slice consists of the models on the LHS of Fig. 1. The aspect model slice is an adaptation of the models on the RHS of Fig. 1. The aspect models must be put into MATA syntax so that they define the failure handling behavior as an increment over the base model slice. Figure 12 therefore shows the state-dependent part of the aspect model slice for failure handling. To make it easier to read, elements that are created or deleted are in bold italics. Note that a MATA rule contains the pattern to match against, the aspect model elements, and the composition operators that detail how those aspect elements are merged with the base. The effect of applying this rule is that: (1) a match is found in the base model with the state diagram sequence pointcut, and (2) the matched submodel of the base is modified by creating and deleting elements according to the <<create>> and <<delete>> composition operators. Note that a combination of <<create>> and <<delete>> is used to move the actions that match against $|X^+$.

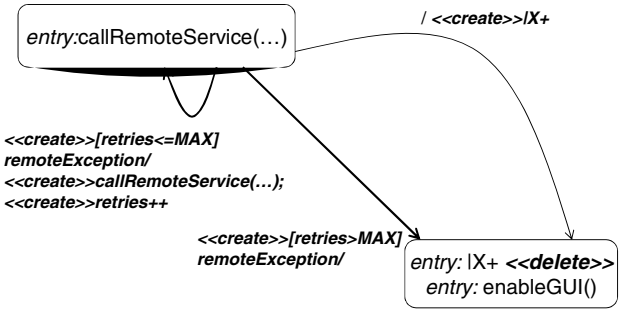

Fig. 12. MATA Specification of the Example in Figure 1

5 Detecting Aspect Interactions

Since aspect models are represented as graph rules in MATA, critical pair analysis can be applied, as explained in Sect. 3, to detect interactions between aspects. In this section, we introduce a small example to illustrate how this works. The example is for class diagrams, but the same principles apply to sequence and state diagrams.

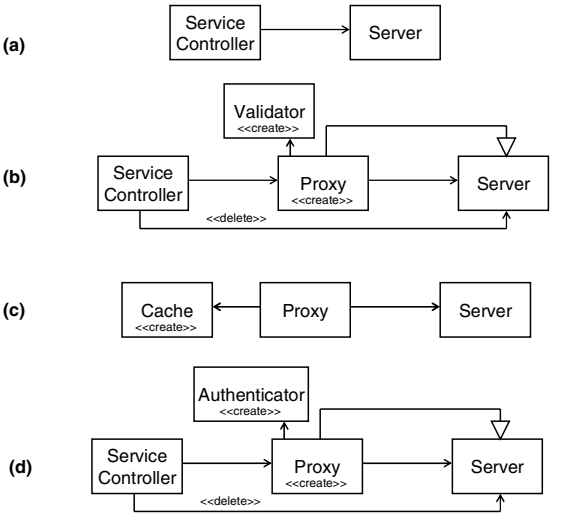

Fig. 13. Simple Example of Aspect Model Interaction

(Note how the concept of immediate neighbor is used so that, for example, <<create>> does not need to be applied to the association from Proxy to Cache.)

Recall the ongoing example, which involves the call of a remote service from a *ServiceController* to a *Server*. Figure 13(a) gives a simple class diagram illustrating the relationship between *ServiceController* and *Server*. Figure 13 (b)-(d) shows three aspects that might be specified to add functionality to the network communication. Figure 13(b) introduces a basic proxy server that simply validates a request before forwarding it. Figure 13(c) is an aspect introducing caching to an existing proxy, and Fig. 13(d) adds an access control proxy. The intention is that all three of these aspects will be added to the base so that all communication between the *ServiceController* and *Server* goes through a caching, validating, access control proxy.

Following the process to use MATA outlined in Sect. 4.1, the modeler would instruct MATA to apply all three aspects and, before actually composing the models, it would apply critical pair analysis to detect dependencies and conflicts between the aspects. Because of the simplicity of the example, it is easy to see in this case that there are indeed serious aspect interactions and that a random order of application of the aspects may result in an incorrect result. For example, if aspect 13(d) is applied to the base first, then aspect 13(b) can no longer be applied because it cannot match the result obtained after applying aspect 13(d)–aspect 13(d) removes the association between *ServiceController* and *Server*, which is needed to match and apply aspect 13(b). Aspect 13(c) will still apply but the result of applying the aspects in this order means that, since 13(b) cannot be invoked, the proxy validity check will not occur. For large examples, such details could easily be overlooked, resulting in incorrect models as a result of applying aspects.

Table 2 summarizes the results of critical pair analysis applied to this example. The table tells us that there is conflict from aspect 13(d) to aspect 13(b). In particular, this means that if aspect 13(d) is applied, then aspect 13(b) cannot be. This matches the intuition in the previous paragraph. Conflicts are generally more serious than dependencies. Dependencies can be dealt with by applying the aspects in a particular order (and this can be specified in the MATA tool). Conflicts, on the other hand, can sometimes be resolved by enforcing an application order, but, in the worst case, imply a fundamental inconsistency in the specification that should be fixed.

Table 2. Dependencies and Conflicts in Figure 13. An entry for row X and column Y implies a dependency or conflict from X to Y.

row→column	**Aspect (b)**	**Aspect (c)**	**Aspect (d)**
Aspect (b)		Dependency	Conflict
Aspect (c)			
Aspect (d)	Conflict	Dependency	

Table 3. Revised Dependencies and Conflicts

row→column	**Aspect (b)**	**Aspect (c)**	**Aspect (d)**
Aspect (b)		Dependency	Dependency
Aspect (c)			
Aspect (d)			

For this example, the modeler might realize, based on the results in Table 2, that a better model would allow aspect 13(b) to introduce the basic validating proxy and then other aspects should add functionality layers on top of this proxy. This would result in modifying aspect 13(d) to only introduce the *Authenticator*. (It would look identical to Fig. 13(c) except *Authenticator* would replace *Cache*.) Once this is done, and critical pair analysis is re-run, the results in Table 3 are obtained. Table 3 shows us that aspects 13(c) and 13(d) are now orthogonal since there are neither dependencies nor conflicts between them. This implies that the application order of 13(c) and 13(d) is irrelevant. However, there are still dependencies from aspect 13(b) to the other rules and so aspect 13(b) must be applied before those. The modeler should therefore specify to apply 13(b) first followed by either 13(c) or 13(d).

6 Extended Example

The preceding sections have introduced the major concepts in MATA. To bring everything together, this section provides an extended example of MATA that includes both static and dynamic models. A cell phone application is used to illustrate the concepts that have been introduced so far.

We will model three use cases for a simple cell phone—*Receive a Call*, *Take a Message*, and *Notify Call Waiting*. The goal here is to compose models for the three use cases. To do this, we will consider *Receive a Call* to be the base use case, and the other two use cases to be aspects. The base use case is modeled in UML, whereas the aspect use cases are modeled as MATA models, that is, as increments of the base models. Note that the models for the aspect use cases refer only to those elements in the base that are needed for the modifications to take place.

Figure 14 shows (simplified) static and dynamic models for the base use case, *Receive a Call*. The phone contains a ringer, a phone component, a display unit, and a keypad. Upon receiving an incoming call, the phone notifies the user by displaying the caller information on the display unit and sending a ring message to the ringer. The user is allowed to either accept the call (then hang up later) or not accept (i.e. disconnect) the call.

Figure 15 gives the behavior models for the two aspects: *Take a Message* and *Notify Call Waiting*. Figure 15(a) is a sequence diagram for *Take a Message*. If the phone rings for a specified amount of time (i.e. there is a timeout), the call goes to a messaging system. In MATA, this is specified by creating a new **alt** fragment since forwarding to voice mail is an alternative scenario to the case where the callee accepts the call. Note that an **any** fragment is used to match against all messages coming after *Ring* in the base. This is needed since once a message is taken, the user should not be able to pick up the call or disconnect it. Hence, the **alt** fragment must be wrapped around all messages in the base concerned with call pick up or disconnect.

In Fig. 15(b), the aspect rule matches any two states that have a transition between them with an event named *Incoming call*. The effect of the aspect is to add an additional transition capturing the voicemail behavior. When this rule is applied, the two states will match against *Idle* and *Waiting* in Fig. 14(c). The effect is to add a transition from *Waiting* back to *Idle*.

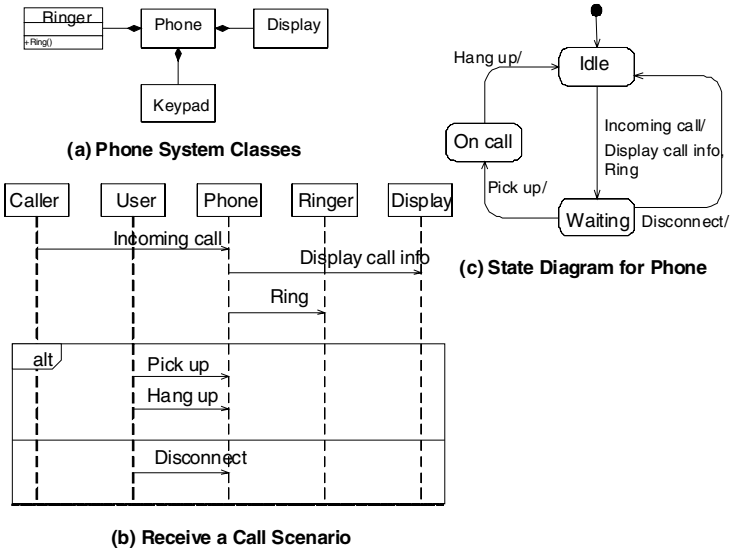

Fig. 14. Models for the Base Use Case

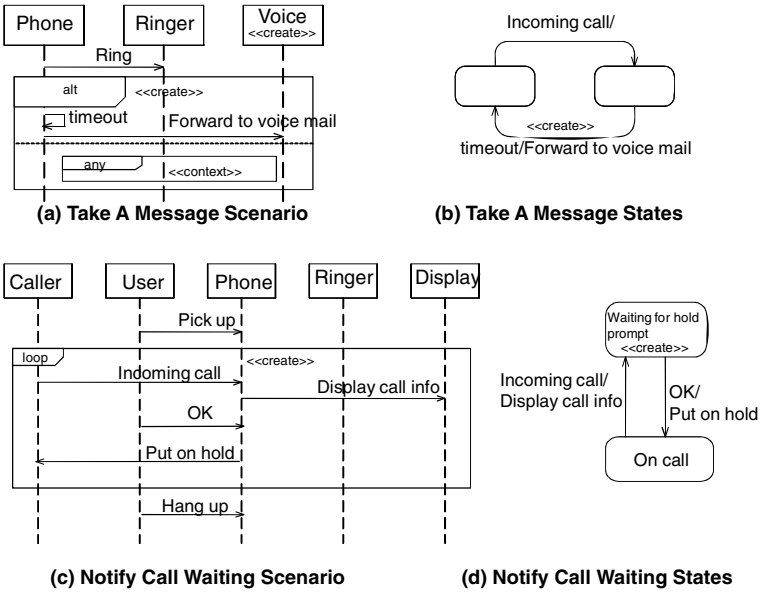

Fig. 15. Aspect Models for Take a Message and Notify Call Waiting

Figure 15(c) introduces messages for putting an incoming call on hold when a call is already underway. These new messages are only relevant when a call is taking place, that is, in between messages *Pick Up* and *Hang Up* in the base. Hence, the **loop** fragment is marked with a <<create>> stereotype and this fragment is inserted in between *Pick Up* and *Hang Up*. Note that, in this case, it would be sufficient to leave out the *Hang Up* message in 15(c), which, in effect, would insert the new behavior *after Pick Up*. However, we include *Hang Up* because there may eventually be other occurrences of *Pick Up,* which should not be affected by the aspect.

Figure 15(d) introduces a new state, *Waiting for hold prompt*, into the base to capture the new behavior for the call waiting use case. Note that the two transitions in 15(d) implicitly have <<create>> stereotypes because they are immediate neighbors of the newly created state.

6.1 Interactions between Aspects

We can see that there is a dependency between the two state diagram rules for *Take a Message* and *Notify Call Waiting*. This dependency arises because *Notify Call Waiting* creates a transition with event *Incoming Call* (Fig. 15(d)) whereas *Take a Message* matches against the event *Incoming Call* (Fig. 15(b)). Hence, if *Take a Message* is applied to the base before *Notify Call Waiting* then any incoming call that is received during an existing call cannot be sent to voicemail. Figure 16 gives the results of composing the two aspects with the base in either order. In 16(a), *Take a Message* is applied to the base before *Notify Call Waiting*. In 16(b), it is applied after. The difference is that there is an extra transition from *Waiting for hold prompt* to *On call* in 16(b) which captures the fact that an incoming call may be sent to voice mail even when there is currently an active call taking place. The difference in the composed state diagrams arises because the rule for *Notify Call Waiting* introduces a new transition with event *Incoming call*. Hence, when the *Take a Message* rule is applied in 16(b), there are two transitions with event *Incoming call* and so the rule applies twice.

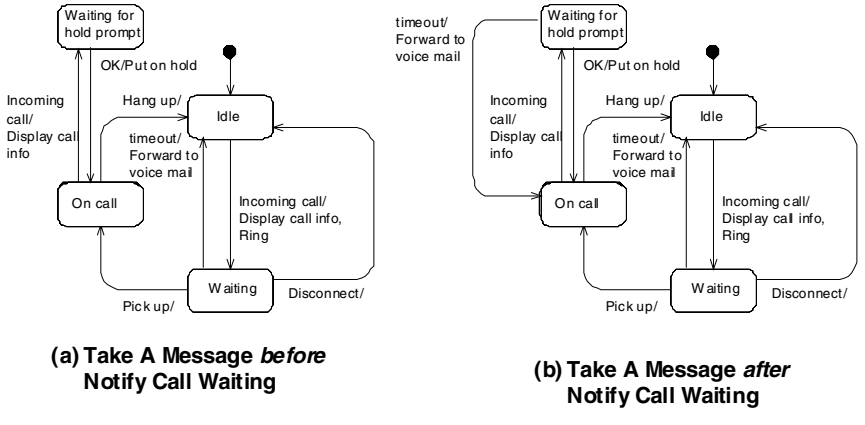

Fig. 16. Base and Aspect State Diagrams Composed

MATA detects these kinds of dependencies automatically. Ultimately, the modeler must decide which order is the correct one, but MATA can at least provide some assistance in flagging cases that must be considered more carefully. If there are no conflicts or dependencies, then the rules can be applied in any order. Critical pair analysis is particularly important when aspects are reused in a different context than originally intended since new conflicts and dependencies may then arise inadvertently.

7 Tool Support

7.1 Overview

This section describes the implementation of the MATA tool. MATA is designed as a vendor-independent tool but currently works on top of IBM's Rational Software Modeler (RSM). Each model slice is modeled as a package. Within this package, the class diagrams, sequence diagrams, and state diagrams for the slice are maintained. A simple UML profile is applied so that the base model slice is stereotyped as <<base>> and aspect model slices are stereotyped as <<aspect>>. Users may select a subset of the aspects and the tool generates the composed model for all of these aspects and the base. The user may also define an ordering of aspect composition in case one aspect needs to be composed before another. If an ordering is not specified, the tool selects an order non-deterministically. Critical pair analysis is always applied before composition and the results are presented to the user.

Since MATA uses graph transformations as the underlying theory, it relies on an existing graph rule execution tool to apply graph rules. The graph rule execution tool used is AGG [21]. MATA converts a UML base model slice, captured as an instance of the UML2 metamodel by RSM, into an instance of a type graph, where the type graph represents a simplified form of the UML2 metamodel. MATA composition rules are converted into AGG graph rules and are executed on the base graph automatically. The results are converted back into a UML2 compliant model and are displayed in RSM. Critical pair analysis is done by AGG and the results are converted into RSM so that detected dependencies and conflicts can be understood by the user.

The details of the conversion to type graphs are not given here. It suffices to say that for simple patterns, the mapping is a straightforward transformation from a UML metamodel instance to a type graph instance. Full details are given in [22]. For sequence pointcuts, the transformation is more complex because AGG does not directly support these. The effect is achieved by tagging model elements to keep track of their relative positioning and then using a sequence of graph rules to manipulate the sequence pointcut matches. This is an implementation detail that we do not go into here. So far, sequence pointcuts have been implemented for sequence diagrams but not for state diagrams.

In principle, MATA could use any existing graph rule execution tool (e.g. VIATRA2 or FUJABA) as its underlying engine, but AGG was chosen because of its support for critical pair analysis. Although built on top of an existing engine, MATA provides some unique features that make it very suitable for aspect modeling and composition, namely: (1) graph rules are defined graphically using the concrete syntax of UML rather than using metaclasses; (2) MATA supports sequence pointcuts, that is, an aspect may match against a sequence of messages or a sequence of

transitions. This is supported directly in the MATA rule syntax; (3) the stereotype <<context>> is unique to MATA; and (4) dependencies and conflicts between aspects can be detected automatically using critical pair analysis.

7.2 Generating AspectWerkz Code from MATA Models

In general, the user has a choice whether to compose the aspect and base models during modeling or to compose them once code has been generated from them. In the former case, the composed models can be used to generate code using existing code generators. In the latter case, aspect-oriented code is generated automatically using MATA's built-in generator, which generates AspectWerkz [18] code. AspectWerkz was chosen for its dynamic weaving capabilities[2] since this research has been conducted within the context of a larger project on integrating model-driven development and runtime weaving. The code generator, however, aims to decouple the MATA representation from the particular AOP language used, and therefore, introduces an intermediate layer in the mapping. This layer defines a metamodel of common AOP language constructs and can be mapped to different AOP languages supporting those constructs.

The remainder of this section gives a brief introduction to AspectWerkz, a short description of the code generation facilities in MATA, and a short example.

7.2.1 AspectWerkz

AspectWerkz is a Java-based AOP language that does not add any new language constructs to Java, but instead supports declaration of aspects via Java annotations. AspectWerkz has now been merged with AspectJ. However, the full dynamic weaving capabilities of AspectWerkz are not available in AspectJ and so we continue to use AspectWerkz in this paper. AspectWerkz includes support for dynamic weaving of aspects, which makes it possible to redefine advices and introductions at runtime without any class reloading or new weaving phase as well as to declare new pointcuts at run time. AspectWerkz was chosen to be the target of MATA's code generator because of its ability to support research projects in adaptive systems. However, because of the merge of AspectWerkz and AspectJ, it would be straightforward to adapt the code generator to produce AspectJ code (albeit without the runtime weaving capabilities). An alternative AOP language with run time weaving facilities would be PROSE [23]. Partly because of the uncertainty of future runtime weaving languages, MATA's code generator has been implemented following MDA principles, that is, by mapping first to an intermediate platform-independent aspect metamodel before mapping to AspectWerkz.

In AspectWerkz, annotations can be used to define aspects (see Fig. 17). An aspect is just a class with the annotation *@Aspect*. The usual advices–before, after and around–can also be defined using annotations. For example, in Fig. 17(b), an around advice is defined to add new behavior to *method1* when *field1* is set to *1*. Introductions in AspectWerkz can be defined using mixins. Figure 17(c) shows a mixin for adding new fields and methods to *Class1*. Note how the mixin is just a class with an annotation.

[2] Although Aspectwerkz has now been integrated into AspectJ 5, the runtime weaving capabilities do not exist in AspectJ 5.

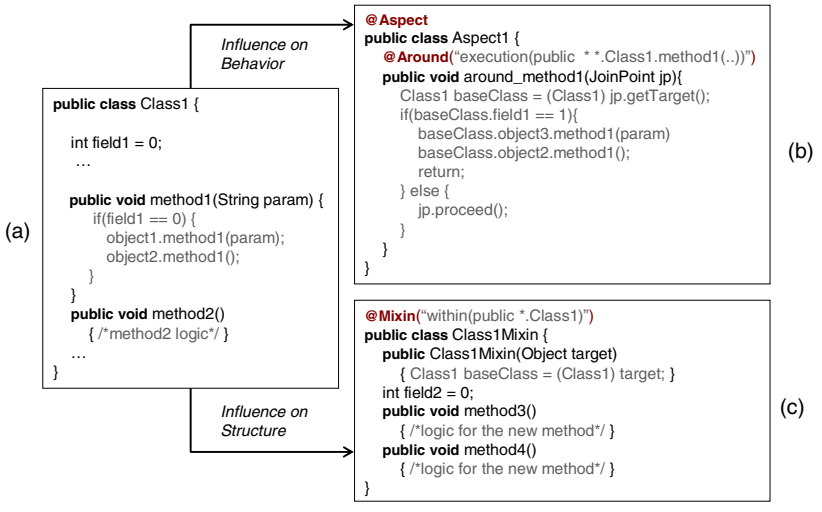

Fig. 17. Syntax of AspectWerkz

7.2.2 Code Generation in MATA

MATA currently generates AspectWerkz code from UML class diagrams and UML state diagrams. It takes a base model slice and a set of aspect model slices (selected by the user) and generates Java code for the base model slice and an AspectWerkz aspect for each of the aspect model slices. State diagrams are implemented using the State pattern.

To maintain independence from the target AOP language, code is generated in two phases. The first phase maps MATA models to an AOP metamodel that defines the concepts common to the most widely used AOP languages but does not commit to a particular AOP language. The second phase generates AspectWerkz code from this metamodel but could be adapted fairly easily to generate, for example, AspectJ code.

The intricacies of the code generator are outside the scope of this paper. Instead, we present a simple example. Recall the cell phone example from Sect. 6. Figure 14 shows the base state diagram, whereas Fig. 15 shows an aspect state diagram that introduced a new state and transitions for the Notify Call Waiting use case.

Figure 18 gives the code generated for these two state diagrams. The LHS of the figure is an implementation of the base state diagram using the State pattern. The RHS uses mixins to add new states and transitions to the base behavior. Note, in this example, that a single new state is created (*Waiting for Hold Prompt)*. This is implemented as a new object that implements the *State* interface. In Fig. 15, a new transition, *Incoming Call*, is added to the *On Call* state. This is captured in the aspect code by a mixin applied to the *OnCall* class. There also needs to be a mixin applied to the *Phone* class to redirect the new transition *OK*. The upper portion of the RHS of Fig. 18 is a bookkeeping code needed to ensure proper placement of the aspect code.

```
public class Caller
   { /* caller interfacing logic */ }
public class User
   { /* user interfacing logic */ }
public class Phone {
   public void incomingCall(String info)
      { curState.incomingCall(info); }
   public void pickUp() { curState.pickUp(); }
   public void hangUp() { curState.hangUp(); }
   public void disconnect() { curState.disconnect(); }

   //State Machine Implementation
   public interface State {
      void incomingCall(String info);
      void pickUp();
      void hangUp();
      void disconnect();
   }
   class Idle implements State {
      public void incomingCall(String info) {
         display.displayCallInfo(info);
         ringer.ring();
         curState = waiting;
      }
      void pickUp() { /*do nothing*/ }
      void hangUp() { /*do nothing*/ }
      void disconnect() { /*do nothing*/ }
   } //other states follow the same approach...
}
public class Ringer() {
   public void ring() { /*ringing logic*/ }
}
public class Display() {
   public void displayCallInfo(String info)
      { /*display logic*/ }
}
```

```
@Aspect
public class NotifyCallWaiting extends MAspect {
   @Around("execution(public  * ReceiveACall.Caller.*(..))"
         +" || execution(public  * ReceiveACall.Phone.*(..))"
         + … /*all base classes referenced by the aspect*/)
   public void crosscut(JoinPoint jp) {
      if(enabled == true)
         this.weave(jp);
      else
         jp.proceed();
   }//…
```

```
@Mixin("within(public * ReceiveACall.Phone)")
public class Phone extends PeerClass {
   //initialization code…
      public void incomingCall(String info) {
            curState.incomingCall(info); }
      public void oK() { curState.oK(); }

   //State Machine Implementation
   interface State
      { void incomingCall(String info); void OK(); }
   @Mixin("within(public ReceiveACall.Phone$OnCall)")
   class OnCall extends PeerState implements State {
      void incomingCall(String info){
         display.displayCallInfo(info);
         setCurState(waitingForHoldPrompt);
      }
      void OK() {/*do nothing*/}
   }
   class WaitingForHoldPrompt extends PeerState implements State {
      void OK(){
         caller.putOnHold();     //instance of NotifyCallWaiting.Caller
         setCurState(OnCall);
      }
      void incomingCall(String info) {/*do nothing*/ }
} // other mixins follow the same approach…
```

Fig. 18. Code Generated for the Cell Phone Example

8 Evaluation and Discussion

This section presents a preliminary evaluation of MATA. In [24], the authors argue that an aspect composition language should satisfy a number of basic requirements. (The arguments made in [24] specifically address aspect-oriented requirements engineering but the discussion generalizes to modeling). We include five of these requirements here and assess whether MATA satisfies them. According to [24], an aspect composition language should aim to be:

1) **Environment-friendly.** A composition language should allow an aspect to be defined without requiring changes to the base model. In particular, the base should not need to be structured or designed in a particular way to support the aspect. This is a special case of obliviousness. If a composition language is very limited in expressiveness, for example, it might require the base to be structured in a particular way. The base would still be oblivious to the aspect, in the sense that it does not expose any aspect-specific interfaces, but the composition could only take place under certain design restrictions applied to the base. In the same way, an aspect should not need to be written in a special way so that it can be composed with the base.

2) **Scalable.** A composition language should scale to large industrial models.
3) **Familiar.** In order to ease adoption of the composition language, it should already be familiar to model developers.
4) **Formal.** The composition technique should be as formal as possible without the formalism becoming a barrier in practice.
5) **Exhaustive.** Models may be composed in many different, complex, and unexpected ways. A composition technique must be exhaustive in that it should provide the means to express all desired compositions. For example, for composing sequence diagrams, composition rules should cover not just sequences and alternatives (i.e. before/after/around) but also concurrency, loops, and interleaving.

We now assess how MATA performs against these criteria. We will focus in this paper on exhaustiveness and will present the results of a small empirical study that suggest that (1) MATA is more exhaustive than competing approaches and (2) that exhaustiveness is required in practical examples. First, however, we will briefly discuss the other requirements. Scientific studies have not yet been undertaken for these.

8.1 Environment-Friendliness

Regarding the first requirement, MATA clearly satisfies it because MATA allows any change to the base model. Hence, any design decisions in the base could ultimately be modified. This is in contrast to other approaches in which only a selection of predefined model elements are allowed to be joinpoints. Therefore, it might be difficult or impossible to modify base elements not in this predefined selection. In Sect. 2, we saw an example where approaches based on AspectJ might be able to define a composition but would do so in a non-optimal way because either the aspect or the base model would have to be broken into fragments, that is, they would have to be written in a particular way to support the composition. The treatment of this example using MATA does not require such decomposition.

As noted above, this criterion is a special case of obliviousness. Recently, a number of authors [25, 26] have argued that full obliviousness is not desirable and that programs should have well-defined interfaces for aspect composition (e.g. joinpoint interfaces). While this argument does not negate the points made in the previous paragraph, we broadly agree with this way of thinking and note that MATA could easily support such interfaces in the future. Currently, all model elements are accessible as joinpoints, but these could potentially be limited by the user. The difference with previous approaches would be that the modeler, instead of the language designer, would have full control over which joinpoints to limit.

8.2 Scalability

This criterion is always difficult to provide evidence for. We have applied MATA in a variety of settings for reasonably large examples, which tends to suggest, at least initially, that it is straightforward to specify aspects using MATA. The major application areas to which we have applied MATA are as follows:

1. *Modeling Software Product Lines*. Jayaraman et al. [27] report on how MATA was used to model features as aspects in software product line development. Each feature is represented as a model slice as an increment over other features. Critical pair analysis was applied to detect feature interactions. As part of this work, Jayaraman et al. took an existing product line–namely, the microwave oven product line from Gomaa's book [28]–and modeled it using MATA.
2. *Maintaining the Separation of Use Cases throughout the Modeling Process using the technique in [6].* We conducted an experiment to refactor a number of student design solutions into an aspect-oriented MATA design –see Sect. 8.5 for details.
3. *Modeling Security Requirements as Aspects*. We have applied MATA to the problem of modeling security concerns during requirements engineering. In particular, security use cases were modeled as MATA sequence diagrams and were composed with sequence diagrams for the base use cases. This approach has been conducted on a number of case studies including an electronic voting system [29] and requirements for a positive train control system [30] under consideration by the Federal Railroad Administration.

These case studies lend evidence that MATA can be used in practice. For larger industrial models, there is, of course, an efficiency question regarding both the graph transformation composition mechanism and critical pair analysis. For both of these, MATA relies on AGG's implementation. In our experience, we have found that composition is very efficient. Critical pair analysis, however, can take time. The efficiency depends on the complexity of the metamodel for the diagram being analyzed. For class diagrams, critical pair analysis generally takes only a few seconds. For state diagrams, it can take a few minutes on large examples. For sequence diagrams, it has taken up to one hour in our most complex case study. This is because the interaction metamodel for UML is very complex. In fact, we have made a number of simplifications to the metamodel to allow us to translate it into a type graph in AGG that allows relatively efficient analysis. This does mean that not all of the modeling elements in sequence diagrams are currently supported by MATA. We consider it a future research question to develop an efficient analyzer for large UML models. The work presented here provides evidence that the analysis would be useful but further work is required on a more efficient implementation. In particular, critical pair analysis in AGG is a very general implementation and it may be that it can be specialized for the specific tasks that MATA takes care of, meaning that the efficiency could be improved.

8.3 Familiarity

For MATA, familiarity means that the MATA language should be as close to UML as possible. Graph transformations are traditionally written over the abstract syntax of a modeling language because this is the most general approach. However, in MATA, aspects (which are graph rules) are written in concrete syntax with a small number of extensions to support sequence pointcuts. The use of UML's concrete syntax makes MATA broadly applicable because no experience with metamodeling is required.

8.4 Formality

Since MATA is based on graph transformations, it is founded on a strong formal footing. The application of critical pair analysis is possible because of this foundation.

8.5 Exhaustiveness

This is the main criterion considered in this paper. As discussed in Sect. 1, there have been two types of approaches to AOM. The first is to use a generic merge algorithm (that can be tailored) to compose an aspect and a base model. The second is to reuse and adapt the joinpoint model and advices from AspectJ. Henceforth, we will refer to the first approach as GM (for generic merge) and the second as AJ.

MATA is more exhaustive than either GM or AJ because any model element can be a joinpoint and any model element can be an advice. However, the question remains whether the additional expressiveness is actually required in practice. To answer this question, we undertook an investigation of existing design solutions to see which kinds of compositions are needed in practice. Our experiment attempted to answer the following question: In practical examples, are model composition mechanisms like GM or AJ enough or is more expressiveness needed? The investigation was undertaken for the use case slice technique of Jacobson and Ng [6]. Use case slices are a way of maintaining a use case-based decomposition throughout the development lifecycle. As an example, for state diagrams, this means that each use case maintains its own state diagram and these state diagrams are composed during late design or implementation to obtain the overall design.

In [6], Jacobson and Ng do not adequately address how to compose use case slices during design. Their approach is to apply AspectJ-like composition operators. The hypothesis of this paper is that such operators are not expressive enough. To test this hypothesis, we examined existing UML designs, refactored those designs to reflect the use case slice technique of Jacobson and Ng, and then investigated the level of expressiveness required to compose designs from different use case slices. Because of the availability of the models, we chose to study seven student team design solutions, each expressed in UML consisting of use cases, class diagrams, interaction diagrams, and state diagrams. Only the use cases and state diagrams were considered in the study, and we focused on compositions of state diagrams from different use case slices.

Projects were conducted by teams of three to four students. Each of the seven projects tackled the same problem statement using the same set of use cases. The scale of the student solutions is clearly not industrial in size and the results offered here are meant to be just the first step.

Based on an analysis of the compositions required in the state-dependent use case slices, we identified four categories of composition that occurred.

C1: One-to-One State Matching. The first category includes model compositions that can be expressed using simple matching of states. In other words, for two state diagrams, $S1$ and $S2$, with state sets $\Sigma 1$ and $\Sigma 2$, the composed state diagram $S1 \bullet S2$, can be obtained by defining a one-to-one mapping $\theta: \Sigma 1 \rightarrow \Sigma 2$. Figure 19(a) gives an example. In the student solutions, this case occurred typically when two use cases defined state diagrams that were joined together into a loop.

C2: Many-to-many state matching. This category is an extension of the previous one whereby states in the two state diagrams have a many-to-many relationship, i.e. $\theta(\sigma)$ is a set for any state σ. This allows a much richer form of composition. In particular, it allows for the creation of composite states (see Fig. 19(b)).
C3: State diagram refactoring. In this category, one or more of the state diagrams must be refactored to enable composition to take place. In other words, one state diagram cannot be inserted in its entirety into the other. Rather, it must be broken up before being inserted in multiple places. This type cannot be handled by state matching because matching cannot refactor a state diagram. Figure 19(c) illustrates this.
C4: State diagram refinement. In this type of composition, additional behavior (i.e. states and transitions) must be added when composition takes place. Clearly, state matching does not apply because state matching cannot refine behavior. This type of composition is necessary in cases where two use case slices have been developed independently but where there are dependencies between the slices that must be resolved when the slices are composed. A typical example concerns access to data. If a single use case slice reads from a data object, then no data access synchronization is required. However, if another use case slice writes to this data object, when the two use case slices are composed, an access synchronization mechanism such as mutual exclusion must be added. Figure 19(d) gives an example.

Based on the student design solutions, we found that all four categories of composition occur for use case slice development. The relative frequency for the four categories was as follows: 13%, 39%, 46% and 2%.

The GM approach supports only category C1 although it can be easily extended to support C2 (as was done in [31]). It does not support categories C3 and C4.
The AJ approach does not support C2 since, for example, composite states cannot be wrapped around multiple base states simply using before/after/around. The AJ approach partially supports categories C3--C4. In some cases, a composition of these types requires container model elements to be wrapped around existing elements–see Fig. 19(d), for instance. AJ does not support this. In some cases, especially for

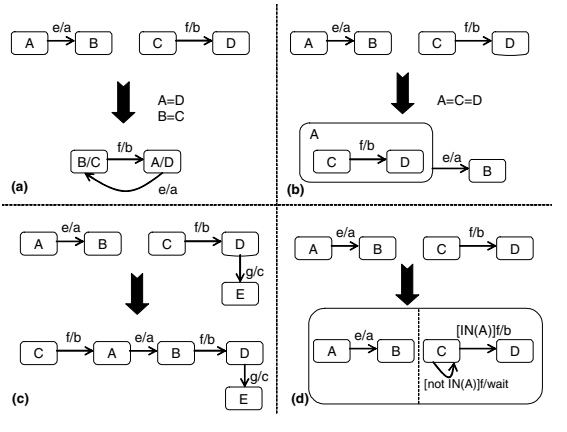

Fig. 19. Composition Categories

category C1, quite complex compositions occur that could be specified by AJ, but the aspect would have to be first refactored into multiple fragments, each of which is then inserted at a different place in the base. We view this as a non-optimal approach to composition because it involves representing fragments of an aspect model separately, which leads to problems in reusability and readability. Finally, in other cases, AJ cannot make a distinction between different kinds of composition. As an example, in Fig. 19(c), inserting the LHS state diagram **after** *f/b* could have two possible results: either stay in state *B* or go to state *D*. With AJ, it is not generally possible to make such a distinction.

MATA supports all categories because the entire state machine diagram syntax is available. For example, two use case slices can be merged in parallel using UML orthogonal regions. The results of the investigation reveal that, at least for use case slice composition, a greater degree of expressiveness is required in practice. Further investigation is required, of course, to see if these results are true for other aspect-oriented software development methods.

9 Related Work

There is a large body of work on AOM, although much of this has been restricted to structural models. Work of note that considers behavioral models is the Motorola WEAVR tool for state machines [5], Song et al.'s work on weaving security aspects into sequence diagrams [32], and Klein et al.'s work on semantic composition for interaction diagrams [33]. The WEAVR tool considers actions in state machines as joinpoints and uses "around" advices to weave in aspect state machines. WEAVR is the first commercially available aspect modeling tool but focuses only on state machines. In addition, it is tailored toward SDL state machines and concentrates on executable modeling and so is more suited to detailed design rather than earlier analysis and design phases.

There has been some work that composes aspect sequence diagrams. Song et al. work [32] has only a very limited set of composition operators and does not provide tool support. However, it does address how to verify the result of the composition by annotating models with OCL expressions, which could then be checked against the composed models. However, the work appears to be in its early stages. Reddy et al. [34] compose aspect sequence diagrams by using special tags that allow an aspect to be broken into pieces and then inserted at different points in the base–for example, at the beginning, in the middle, or at the end of the base messages. Whilst interesting, the MATA approach is more general and subsumes these operators. Indeed, earlier work by some of the authors of this paper considered composition of sequence diagrams using a limited set of composition operators [35]. This work has also been subsumed by MATA. Klein and Kienzle [36] describe a case study of composing aspect sequence diagrams. In this approach, one sequence diagram describes the pointcut and another describes the advice. The paper presents a case study using the semantic composition of scenarios described in [33]. The latter is important work that goes beyond syntactic mechanisms for defining pointcuts but instead relies on the semantics of the modeling language for matching an aspect. This reduces, to some extent, the fragile pointcut problem for aspect sequence diagrams but does incur a performance overhead. Such techniques could potentially be incorporated into MATA.

Other work on AOM includes, of course, Theme/UML [3]. Theme/UML is an example of the generic matching approach considered in Sect. 8 and suffers the limitations in expressiveness noted there. Katara and Katz [37] provide an approach for AOM of sequence and state diagrams based on superimposition. This is quite similar to MATA in that aspects are defined as increments over other models (either the base or other aspects). However, Katara and Katz [37] does not support a fully-fledged pattern language for defining pointcuts, which limits the quantification possible. Although Katara and Katz do give consideration to identifying dependencies between aspects, these dependencies must be found manually and documented on a so-called concern diagram. Indeed, MATA can be thought of as providing automated support for developing and/or validating such a concern diagram.

Generic aspects can be seen as a kind of design pattern. Hence, work on instantiating design patterns and applying aspect models is closely related. Indeed, there has been some work on automatically instantiating generic descriptions of design patterns [38, 39] and using such techniques in AOM [31, 40].

MATA views aspect composition simply as model transformation. This is a point of view that has also been noted by others. A general discussion of the similarities of model composition and model transformation is presented in [41]. One interesting point described there, and discussed elsewhere, is that aspect composition could either be specified by a generic model transformation language or by a dedicated aspect composition language, or indeed that there is a spectrum of possibilities lying in between. MATA tends toward the use of a generic model transformation language but tailors this to ensure familiarity of the language to modelers. In this sense, it is different than using a completely general transformation language, such as the one based on QVT, but retains the power and flexibility of a generic transformation language. Dedicated aspect composition languages risk sacrificing expressiveness because a limited number of composition operators would be provided. For example, France et al. [42] provide such a limited number of matching and composition operators but the user may override these if necessary, or indeed define new operators. However, this requires programming skills. MATA brings flexible composition without requiring any knowledge of programming or the need to understand the code in an existing composition framework. France et al. [42] is also limited to class diagrams. It is not clear how these techniques would extend to behavioral models.

MATA provides two key contributions to AOM. First, is the support for detecting aspect interactions. Second, it supports sequence pointcuts. To date, there has been limited support for detecting aspect interactions in AOM. Aspect interactions are a well-recognized problem but research has tended to focus on how to document interactions rather than uncover them automatically (cf. [37, 43--45]). The only known work is [46], which translates aspect UML models into Alloy so that they can be verified. This approach does not consider behavioral diagrams but requires pre/post-conditions to specify operations on class diagrams. Furthermore, it is more of a general verification approach not specifically geared toward interactions. This means that it could potentially uncover more semantic interactions (which MATA cannot) but at the cost of a more expensive analysis. At the programming level, there has been research on detecting interactions using static analysis [47, 48].

Although expressive pointcut mechanisms, such as sequence pointcuts, have been considered for AOP [12], to the authors' knowledge, this paper is the first work to

bring expressive pointcuts to behavioral models. Related work that is closest to ours is joinpoint designation diagrams (JPDDs) [13]. JPDDs are similar to defining patterns using graph rules. Something similar to sequence pointcuts can be defined but the advices are limited to before/after/around. Furthermore, the advantage of using graph rules is the existence of formal analysis techniques. In addition, JPDDs focus on defining joinpoints and are not so much concerned with composition. MATA provides a full composition tool in which very expressive composition relationships can be specified. This is not possible with JPDDs.

This paper considers joinpoints to be static in the sense that the runtime semantics of behavioral diagrams is not taken into consideration. Dynamic joinpoints can also be defined for behavioral models, such as state diagrams [9]. However, since currently models are most commonly used for communication and documentation, and are not necessarily executed, static joinpoints are perhaps more useful in current modeling practices. It would be interesting to extend MATA to dynamic joinpoints, however.

More generally, model composition has been addressed outside of the AOSD community. In particular, [49] investigates how to merge state machines using composition relationships and category theory. This is similar in many respects to our work but has a different goal in that it addresses how to reconcile models produced by different development teams.

10 Conclusion and Further Work

This paper has presented a new approach for AOM wherein aspect composition is considered to be simply a special case of model transformation. A language and tool, MATA, has been presented, which allows modelers to maintain aspect models separately, detect structural interactions between aspects automatically, and compose a chosen set of aspects automatically with a set of base models. The approach goes beyond previous work in that:

- MATA provides a unified approach to aspect model composition. Any modeling language with a well-defined metamodel can be handled in the same way. Currently, UML class, sequence, and state diagrams are supported, but extensions to other modeling languages would be straightforward and would provide the same capabilities in detecting interactions and automating composition.
- MATA provides a richer aspect composition language. Joinpoints are defined by an expressive pattern language and any base model element (or combination of elements) can be a pointcut. In particular, MATA provides the first full support for sequence pointcuts at the aspect modeling level.

MATA is supported by a tool built on top of IBM's Rational Software Modeler. It has been applied in a range of application areas, including security modeling, software product lines, and modeling of use case slices.

There are a number of interesting avenues for further work that would build upon MATA. First, base models in MATA are currently completely open, in the sense that any base model elements can be accessed by aspect models. This has shown to be absolutely essential in some application areas. In particular, for the software product line

method PLUS [28], which can be handled in MATA by modeling features as aspects, models of non-kernel features can be added to models of the kernel in many and varied different ways. It would not have been possible to restrict the joinpoint model and still allow the case studies from [28] to be modeled faithfully.

However, it may be desirable for other application areas to restrict the joinpoint model so that only certain base model elements can be affected by an aspect. This kind of approach would potentially support improved modular reasoning for aspects. MATA could support such a technique easily as interfaces could be designed on top of the existing language. In any case, we feel that the modeler should be in control of whether or not full access is required by the aspects and it is not up to the language designer to restrict the joinpoint model for him/her.

Another area where MATA could potentially be extended is to provide domain-specific composition operators, built on top of the existing language. A key contribution of this paper is that MATA allows all modeling languages to be handled in a uniform way. However, the current composition operators in MATA are quite low level because they are at the same level as the underlying modeling language. One could imagine defining more abstract operators, for example, in software architecture composition that would be then mapped down to MATA's operators. This would raise the level of discourse of aspect modelers but would retain the strong benefits of the MATA foundations. However, such a path should be taken with caution. A great deal of effort has already gone into language design for existing modeling languages and it is not completely clear that an additional layer of abstraction would be beneficial.

Along similar grounds, MATA's composition is purely syntactic currently. This means that aspect modelers define aspects based on the syntactic elements of the underlying modeling language. While this is in line with current practice in modeling, it would be interesting to investigate semantics-based composition techniques, similar to those developed for aspect-oriented requirements engineering languages [50]. This would allow modelers to specify aspects in terms of semantic concepts of the domain rather than syntactic modeling elements. For example, one might wish to define the pointcut of all model elements related to access control. The techniques in [50] rely on natural language processing techniques to extract semantic content from textual requirements documents and it is not clear how such an approach could be adapted to analysis and design models. However, it is certainly an open area of research that could provide fruitful solutions to the fragile pointcut problem in AOM.

The usability of the MATA composition language has not yet been fully tested. Although a number of realistic case studies have been undertaken, we have limited experience with real users. The intricacies of the propagation algorithm are such that it may be difficult to grasp for novices. However, use of propagation is always optional and the user may choose to explicitly provide stereotypes. So far, MATA provides no support for validating the composition of base and aspect. It is possible to get unexpected results if there are interactions between aspects that cannot be detected by critical pair analysis. A simple example is if two aspects each create an instance of the same class. Then the result will have two copies of this instance where only one may be desired. There may be lightweight techniques that can help with validating the composition. Another usability issue is in maintaining the generality of the aspects. Generic aspects should be designed where possible so that they can be reused. This is

certainly easy to do in MATA because of the rich pattern-matching facilities. However, from a usability point of view, more research is required as to how to guide users to specify good (i.e. generically applicable) aspects.

One of the main points made in this paper is that aspect composition approaches based on generic match and merge algorithms–for example, those that merge model elements by name–are not very practical. This is a claim backed up by preliminary empirical evidence in Sect. 8.5. On the other hand, there may be some advantages in combining a MATA-like approach with these generic merge algorithms. Once again, this could provide a way of raising the level of composition abstraction in MATA. Care would need to be taken, however, to ensure that the problems of generic merge algorithms–that the results of composition are hard to predict and adapt–do not carry over to the MATA context.

Finally, we hope that the expressive composition mechanisms provided by MATA might have some consequences for AOP. Whilst modeling is different from programming, it seems that AOP could also benefit by more expressive pointcut languages or more expressive advices. We believe that the rich language available in MATA might offer some insights as to how such languages should be developed.

References

[1] Whittle, J., Moreira, A., Araújo, J., Rabbi, R., Jayaraman, P., Elkhodary, A.: An Expressive Aspect Composition Language for UML State Diagrams. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 514–528. Springer, Heidelberg (2007)

[2] Whittle, J., Jayaraman, P.: MATA: A Tool for Aspect-Oriented Modeling based on Graph Transformation. In: Workshop on Aspect Oriented Modeling at the International MODELS Conference, Nashville, TN (2007)

[3] Clarke, S., Baniassad, E.: Aspect-Oriented Analysis and Design: The Theme Approach. Addison Wesley, Reading (2005)

[4] France, R., Ray, I., Georg, G., Ghosh, S.: Aspect-oriented approach to early design modeling. In: IEE Proceedings - Software, vol. 151, pp. 173–186 (2004)

[5] Cottenier, T., van den Berg, A., Elrad, T.: Motorola WEAVR: Model Weav-ing in a Large Industrial Context. In: Aspect-Oriented Software Development (AOSD), Vancouver, Canada (2007)

[6] Jacobson, I., Ng, P.-W.: Aspect Oriented Software Development with Use Cases. Addison-Wesley Professional, Reading (2004)

[7] Reddy, Y.R., Ghosh, S., France, R., Straw, G., Bieman, J., McEachen, N., Song, E., Georg, G.: Directives for Composing Aspect-Oriented Design Class Models. In: Rashid, A., Aksit, M. (eds.) Transactions on Aspect-Oriented Software Development I. LNCS, vol. 3880, pp. 75–105. Springer, Heidelberg (2006)

[8] Fleury, F., Baudry, B., France, R., Ghosh, S.: A Generic Approach for Automatic Model Composition. In: Workshop on Aspect Oriented Modeling at MODELS 2007 (2007)

[9] Zhang, G., Hölzl, M., Knapp, A.: Enhancing UML State Machines with Aspects. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 529–543. Springer, Heidelberg (2007)

[10] Lopez-Herrejon, R., Batory, D.: Modeling Features in Aspect-Based Product Lines with Use Case Slices: An Exploratory Case Study. In: Kühne, T. (ed.) MoDELS 2006. LNCS, vol. 4364, pp. 6–16. Springer, Heidelberg (2007)
[11] Rashid, A.: Views, Aspects and Roles: Symphony or Random Noise? In: Panel Statement at Views, Aspects and Roles Workshop associated with ECOOP 2005 (2005)
[12] Douence, R., Fritz, T., Loriant, N., Menaud, J.-M., Segura-Devillechaise, M., Sudholt, M.: An Expressive Aspect Language for System Applications with Arachne. In: Aspect-Oriented Software Development (AOSD), Chicago, Illinois, pp. 27–38 (2005)
[13] Stein, D., Hanenberg, S., Unland, R.: Expressing Different Conceptual Models of Join Point Selections in Aspect-Oriented Design. In: Aspect-Oriented Software Development (AOSD), Bonn, Germany, pp. 15–26 (2006)
[14] Markovic, S., Baar, T.: Refactoring OCL Annotated UML Class Diagrams. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 280–294. Springer, Heidelberg (2005)
[15] de Micheaux, N.L., Rambaud, C.: Confluence for Graph Transformations. Theoretical Computer Science 154, 329–348 (1996)
[16] Wagner, R.: Developing Model Transformations with Fujaba. In: International Fujaba Days, Bayreuth, Germany, pp. 79–82 (2006)
[17] Balogh, A., Varro, D.: Advanced Model Transformation Language Constructs in the VIATRA2 Framework. In: ACM Symposium on Applied Computing (Model Transformation Track), Dijon, France, pp. 1280–1287 (2006)
[18] Boner, J., Vasseur, A.: Tutorial on AspectWerkz for Dynamic Aspect-Oriented Programming. In: Aspect Oriented Software Development (2004)
[19] Moreira, A., Rashid, A., Araújo, J.: A Multi-Dimensional Separation of Concerns in Requirements Engineering. In: International Conference on Requirements Engineering (RE), Paris, France, pp. 285–296 (2005)
[20] Rudolf, M.: Utilizing Constraint Satisfaction Techniques for Efficient Graph Pattern Matching. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) TAGT 1998. LNCS, vol. 1764, pp. 238–251. Springer, Heidelberg (2000)
[21] Taentzer, G.: AGG: A Graph Transformation Environment for Modeling and Validation of Software. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) AGTIVE 2003. LNCS, vol. 3062, pp. 446–453. Springer, Heidelberg (2004)
[22] Jayaraman, P.: Interaction Verification and Model Composition in Product Lines Using MATA in Dept. of Information and Software Engineering. MS Thesis Fairfax, VA. George Mason University, USA (2007)
[23] Nicoara, A., Alonso, G.: Dynamic AOP with PROSE. In: International Workshop on Adaptive and Self-Managing Enterprise Applications at CAiSE, Porto, Portugal (2005)
[24] Mussbacher, G., Amyot, D., Whittle, J., Weiss, M.: Flexible and Expressive Composition Rules with Aspect-Oriented Use Case Maps (AoUCM). In: Moreira, A., Grundy, J. (eds.) Early Aspects Workshop 2007 and EACSL 2007. LNCS, vol. 4765, pp. 19–38. Springer, Heidelberg (2007)
[25] Griswold, W., Sullivan, K., Song, Y., Shonle, M., Tewari, N., Cai, Y., Rajan, H.: Modular Software Design with Crosscutting Interfaces. IEEE Software 23, 51–60 (2006)
[26] Aldrich, J.: Open Modules: Modular Reasoning about Advice. In: Black, A.P. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 144–168. Springer, Heidelberg (2005)
[27] Jayaraman, P., Whittle, J., Elkhodary, A., Gomaa, H.: Model Composition in Product Lines and Feature Interaction Detection using Critical Pair Analysis. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 151–165. Springer, Heidelberg (2007)

[28] Gomaa, H.: Designing Software Product Lines with UML: From Use Cases to Pattern-based Software Architectures. Addison-Wesley Object Technology Series (2005)
[29] Kohno, T., Stubblefield, A., Rubin, A., Wallach, D.: Analysis of an Electronic Voting System. In: IEEE Symposium on Security and Privacy, pp. 27–40. IEEE Computer Society Press, Los Alamitos (2004)
[30] Hartong, M., Goel, R., Wijesekera, D.: Use Misuse Case Driven Forensic Analysis of Positive Train Control: A Preliminary Study. In: 2nd IFIP WG 11.9 International Conference on Digital Forensics, Orlando, FL
[31] Araújo, J., Whittle, J., Kim, D.-K.: Modeling and Composing Scenario-Based Requirements with Aspects. In: International Conference on Requirements Engineering, Kyoto, Japan, pp. 58–67 (2004)
[32] Song, E., Reddy, R., France, R.B., Ray, I., Georg, G., Alexander, R.: Verifiable Composition of Access Control and Application Features. In: ACM Symposium on Access Control Models and Technologies (SACMAT), Stockholm, Sweden, pp. 120–129 (2005)
[33] Klein, J., Helouet, L., Jézéquel, J.-M.: Semantic-Based Weaving of Scenarios. In: Aspect-Oriented Software Development (AOSD), Vancouver, Canada, pp. 27–38 (2006)
[34] Reddy, R., Solberg, A., France, R., Ghosh, S.: Composing Sequence Models Using Tags. In: Aspect Oriented Modeling Workshop at MODELS 2006 (2006)
[35] Whittle, J., Araújo, J.: Scenario Modelling with Aspects. In: IEE Proceedings - Software, August 2004, vol. 151, pp. 157–172 (2004)
[36] Klein, J., Kienzle, J.: Reusable Aspect Models. In: Aspect Oriented Modeling Workshop at MODELS 2007 (2007)
[37] Katara, M., Katz, S.: Architectural Views of Aspects. In: Aspect-Oriented Software Development (AOSD), Boston, Massachusetts, pp. 1–10 (2003)
[38] Kim, D.-K.: Evaluating Conformance of UML Models to Design Patterns. In: International Conference on the Engineering of Complex Computer Systems (ICECCS), Shanghai, China, pp. 30–31 (2005)
[39] Kim, D.-K., Whittle, J.: Generating UML Models from Domain Patterns. In: Software Engineering Research, Management and Applications, pp. 166–173 (2005)
[40] Kim, D.K.: A Pattern-Based Technique for Developing UML Models of Access Control Systems. In: 30th Annual International Computer Software and Applications Conference (COMPSAC), Chicago, IL, pp. 317–324 (2006)
[41] Baudry, B., Fleurey, F., France, R., Reddy, R.: Exploring the Relationship between Model Composition and Model Transformation. In: Aspect Oriented Modeling Workshop at MODELS 2005 (2005)
[42] France, R., Fleurey, F., Reddy, R., Baudry, B., Ghosh, S.: Providing Support for Model Composition in Metamodels. In: IEEE International EDOC Conference, Annapolis, Maryland (2007)
[43] Zhang, J., Cottenier, T., van den Berg, A., Gray, J.: Aspect Interference and Composition in the Motorola Aspect-Oriented Modeling Weaver. In: Aspect Oriented Modeling Workshop at MODELS 2006 (2006)
[44] Sanen, F., Loughran, N., Rashid, A., Nedos, A., Jackson, A., Clarke, S., Truyen, E., Joosen, W.: Classifying and Documenting Aspect Interactions. In: Workshop on Aspects, Components and Patterns for Infrastructure Software at AOSD, Bonn, Germany (2006)
[45] Bakre, S., Elrad, T.: Scenario-based Resolution of Aspect Interactions with Aspect Interaction Charts. In: Aspect-Oriented Software Development, Vancouver, Canada, pp. 1–6 (2007)

[46] Mostefaoui, F., Vachon, J.: Design-level Detection of Interactions in Aspect-UML Models using Alloy. Journal of Object Technology 6, 137–165 (2007)
[47] Douence, R., Fradet, P., Südholt, M.: A Framework for the Detection and Resolution of Aspect Interactions. In: Generative Programming and Component Engineering, Pittsburgh, PA, pp. 173–188 (2002)
[48] Douence, R., Fradet, P., Südholt, M.: Composition, reuse and interaction analysis of stateful aspects. In: Aspect Oriented Software Development, pp. 141–150 (2004)
[49] Nejati, S., Sabetzadeh, M., Chechik, M., Easterbrook, S., Zave, P.: Matching and Merging of Statecharts Specifications. In: International Conference on Software Engineering, pp. 54–64 (2007)
[50] Chitchyan, R., Rashid, A., Rayson, P., Waters, R.: Semantics-Based Com-position for Aspect-Oriented Requirements Engineering. In: Aspect-Oriented Software Development (AOSD), Vancouver, Canada, pp. 36–48 (2007)

Appendix

This appendix describes how MATA performs the conversion from a model in concrete syntax to a type graph in AGG. This conversion process is performed automatically.

MATA considers a subset of the UML metamodel (we do not yet consider the full UML2 metamodel) and maps it to a corresponding type graph. The type graph represents the metamodel in the AGG syntax. In the current scope, the chosen UML metamodel subset contains commonly used modeling elements of class diagrams, sequence diagrams, and state machines. MATA converts a base model into an AGG graph and converts an aspect model into an AGG graph rule.

To illustrate, we present a simple example for a family of printers. A printer will be modeled as the base and an optional feature, a sheet rotator (which allows printing on both sides of a sheet), will be modeled as an aspect.

Class Diagrams

The base model contains an assembly of an abstract controller object called Printer. The Printer aggregates PrintRoller and PrintNozzle objects. Figure 20 shows the class diagram of the Printer base model in concrete UML syntax. The graph metamodel used to represent the class diagram is shown in Fig. 21. The corresponding host graph of the Printer base model is shown in Fig. 22. The class diagram concepts supported by MATA are:

1. Class/Interface–A class or an interface is represented by a node of type Classifier. The Type attribute indicates whether the node is a class or an interface. Additional attributes such as Name and Visibility indicate the name and visibility of the element. The attribute isAbstract is used to represent an abstract class.
    a. Property–A graph node of type Attribute represents properties of classes and interfaces. These nodes are connected to the owning Classifier nodes via an edge of type Owns. The attributes Name,

Visibility, isStatic, Lower and Upper indicate the name, visibility, static nature, lower and upper bound of the attribute, respectively.

b. Operation–A graph node of type Operation represents operations supported by classes and interfaces. An operation node is connected to the owning classifier node via an edge of type Owns. The attributes Name, Visibility, isAbstract, and isStatic indicate the name, visibility, abstract, and static nature of the operation, respectively.

2. Generalization–An edge of type Extends represents the generalization relationship between two classes or interfaces. The edge connects the corresponding nodes of type classifier.
3. Realization–An edge of type Implements represents the realization of an interface by a class. In the graph metamodel, this edge connects a classifier of type interface to a classifier of type class.
4. Association/Composition/Aggregation–An edge of type Association represents a relationship between two classifiers. Table 4 explains the representation of different kinds of relationships such as associations, compositions, and aggregations as well as other association-related attributes.

Fig. 20. Class diagram for Printer Kernel feature

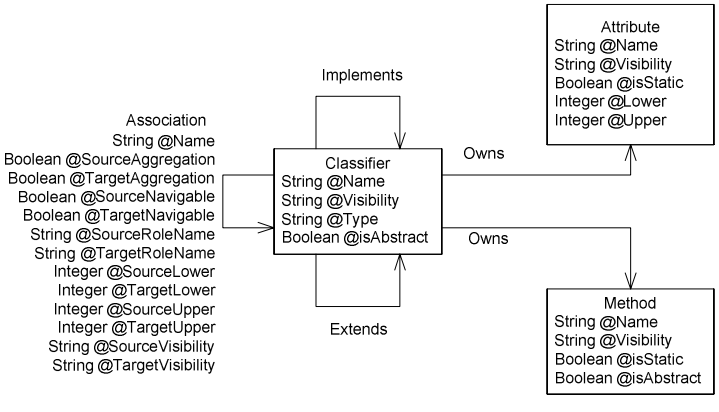

Fig. 21. Graph metamodel for class diagram (in AGG syntax: e.g. String @Name means Name is of type String)

Table 4. Graph metamodel attributes of an Association

Attribute	Description
SourceAggregation	Represents the aggregation kind of the source classifier of the association.
TargetAggregation	Represents the aggregation kind of the target classifier of the association.
SourceRoleName	Represents the name of the source role of the association.
TargetRoleName	Represents the name of the target role of the association.
SourceLower	Represents the lower bound of the source of the association.
TargetLower	Represents the lower bound of the target of the association.
SourceUpper	Represents the upper bound of the source of the association.
TargetUpper	Represents the upper bound of the target of the association.
SourceVisibility	Represents the visibility of the source of the association.
TargetVisibility	Represents the visibility of the target of the association.

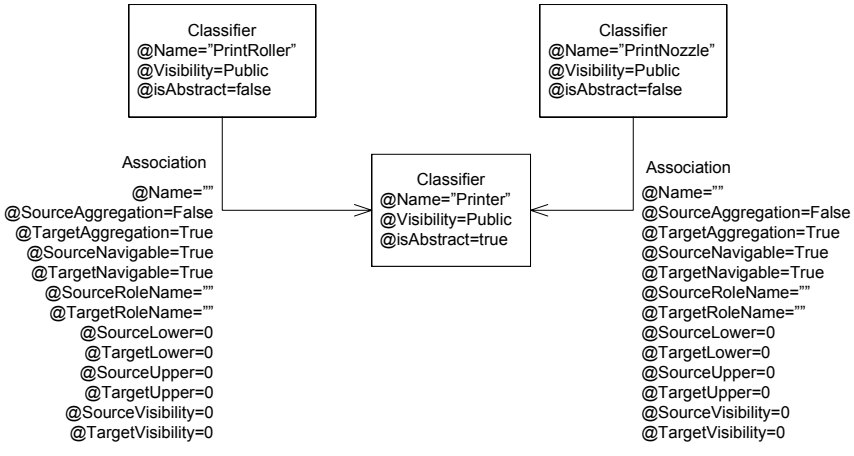

Fig. 22. Host graph for Printer Kernel class diagram

Sequence Diagram

The Printer object receives a print command from an external user and sends a message to the PrintRoller to lift a sheet from an external paper tray. Then, it sends a message to the PrintNozzle to start printing on the sheet and when the sheet is printed, the PrintRoller ejects the sheet. The process repeats if the print job requires more sheets.

Figure 23 shows the sequence diagram of the Printer Kernel in concrete UML syntax. The graph metamodel used to represent the sequence diagram is shown in Fig. 24. The corresponding host graph of the Printer Kernel feature is shown in Fig. 25. The sequence diagram related concepts supported by MATA are

1. Interaction–An interaction of type sequence diagram is represented by a node of type Sequence Diagram.
2. OccurrenceSpecification/GeneralOrdering–An OccurrenceSpecification is represented by a node of type Sequencer. The after association of GeneralOrdering is represented by an edge of type Next between two Sequencer nodes. These nodes are also used to indicate the start and end of interaction diagrams, interaction fragments and interaction operands. For example, the start and end of an interaction are represented individually by two sequencer nodes that are connected to the Sequence Diagram node by edges of start and end type, respectively.
3. Lifeline–The lifeline of a participant in a sequence diagram is represented by a node of type Class. The name of the lifeline is preserved by the Name attribute of the node. MATA does not support explicit creation or destruction of a lifeline and assumes a lifeline to exist throughout the interaction diagram.
4. CombinedFragment–A fragment is represented by a node of type Fragment.
    a. InteractionOperator–The interaction operator of a fragment is preserved by the Operator attribute of the node representing the fragment.
    b. InteractionConstraint–A constraint applied on a fragment is preserved by the Guard attribute of the node representing the fragment.
    c. Interaction operand–Each operand of a fragment is represented by a node of type Operand.
5. Complete Asynchronous Message–Complete asynchronous messages are represented using nodes of type Message. The name of the asynchronous message is preserved by the Name attribute of the Message node. The sending and receiving lifelines of a message are indicated by edges of type Receiver and Sender from the Message node to the class nodes, respectively.
6. EventOccurrence (Send/Receive)–The receive and send events of a message are represented individually by sequencer nodes connected by an edge of type Next.

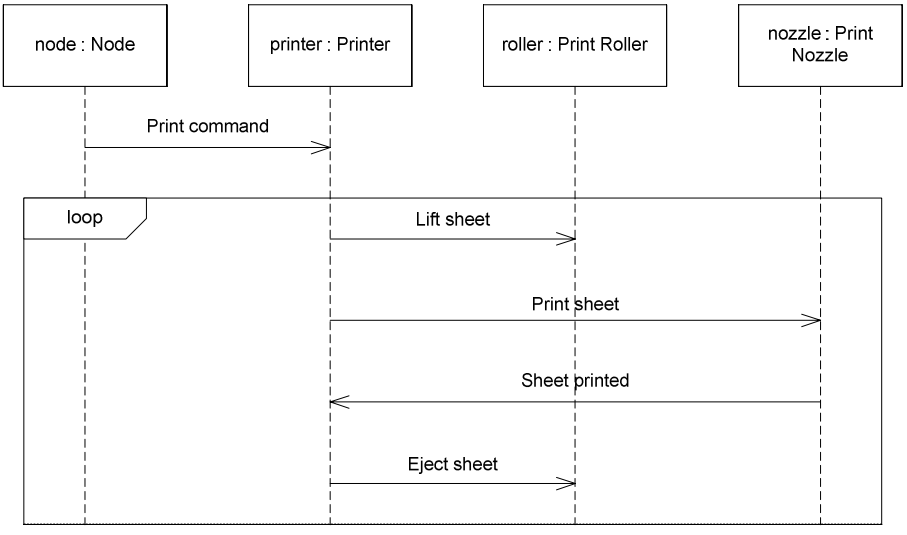

Fig. 23. Sequence diagram for Printer Kernel feature

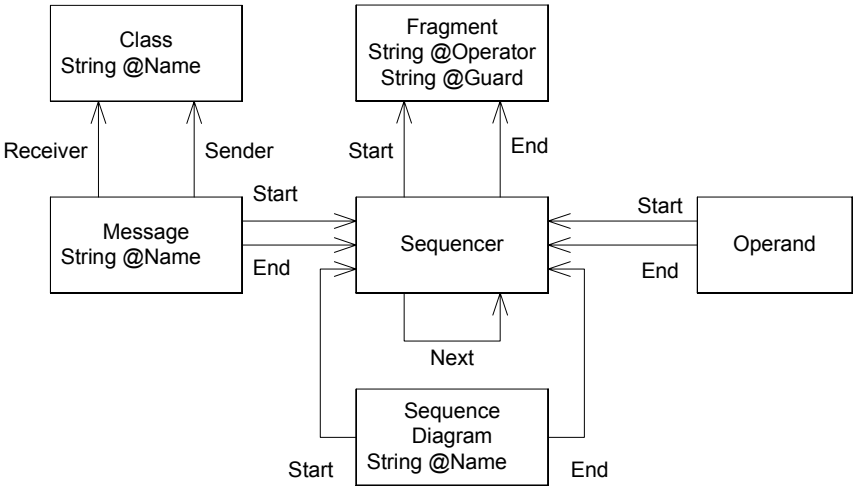

Fig. 24. Graph metamodel for sequence diagram

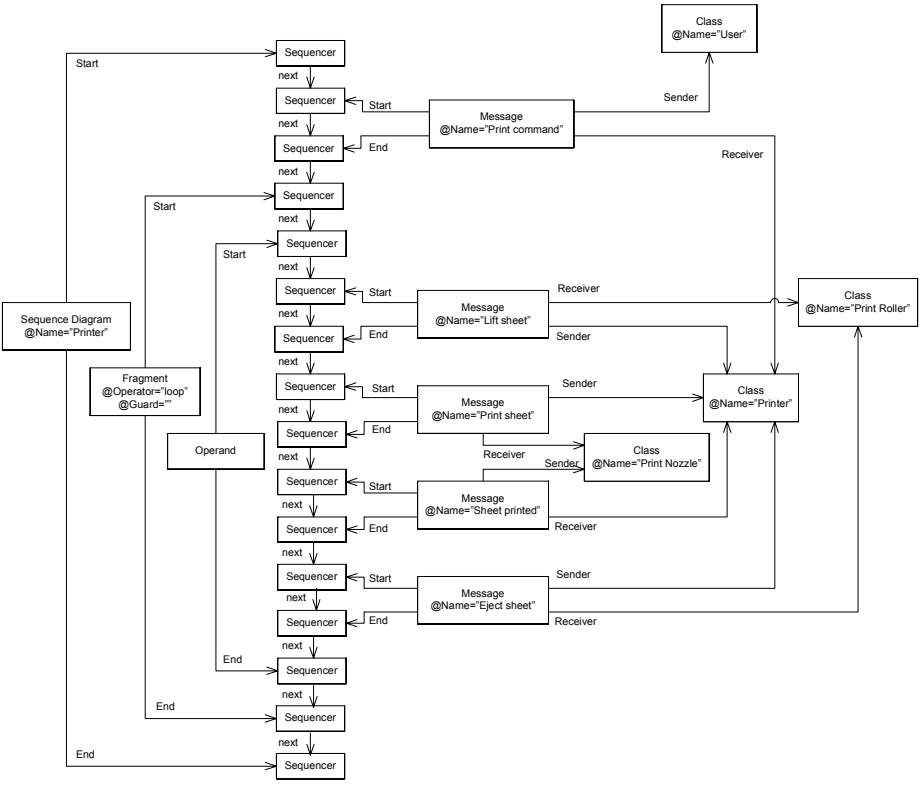

Fig. 25. Host graph of Printer Kernel sequence diagram

MATA Syntax

MATA translates a UML model annotated with the MATA stereotypes to a graph rule. The procedure for generating the graph rules is as follows:

1. Instantiate a graph rule with a left and a right graph.
2. For each element in the source model:

    2.1. If the element is stereotyped with <<create>>, create a graph node and add the node to the right graph.
    2.2. If the element is stereotyped with <<delete>>, create a graph node and add the node to the left graph.
    2.3. If the element is stereotyped with <context>>, create two graph nodes and add one to the left graph and the other to the right graph. Add mapping information between the nodes.
    2.4. If the element is not associated with any stereotype:

2.4.1. If the element is a nearest neighbor of another element in the model then apply the stereotype of the neighbor to the element and repeat step 2. For example, if a class element is stereotyped with <<create>> or <<delete>> then the same stereotype is implicitly applied to all attributes and methods that are owned by the class.

2.4.2. Else, create two graph nodes and add one to the left graph and the other to the right graph. Add mapping information between the nodes.

Sheet rotator Aspect

The sheet rotator aspect adds flip sheet functionality to the PrintRoller object. The static view transformation for this rule, called AddFlipMethod, is shown using concrete syntax and graph syntax in Figs. 26 and 27, respectively.

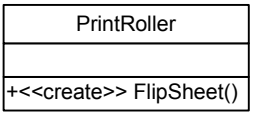

Fig. 26. Concrete syntax for rule AddFlipMethod

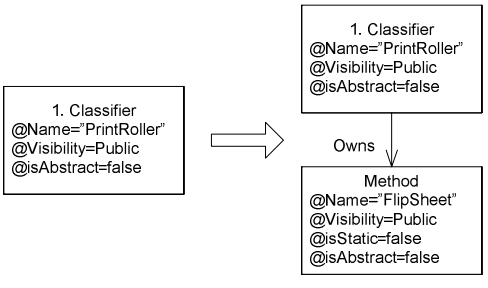

Fig. 27. Graph syntax for rule AddFlipMethod

This functionality is invoked only if one side of the sheet has been printed and the print job requires more sheets. The Printer object adds an alternate flip sheet message to an existing eject sheet message. The printer sends the lift sheet message only if the sheet has been ejected or if the first sheet is being printed. Two separate transformations are used to execute these changes. The first rule to add an alternate flip sheet message is called AddFlipMessage and is shown using UML concrete syntax and graph syntax in Figs. 28 and 29, respectively. The second rule to make the lift sheet message optional is called MakeLiftOptional and is not shown here.

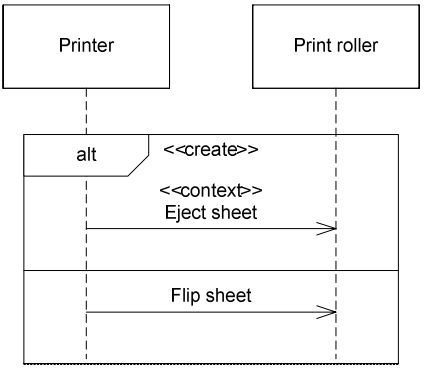

Fig. 28. Concrete syntax for rule AddFlipMessage

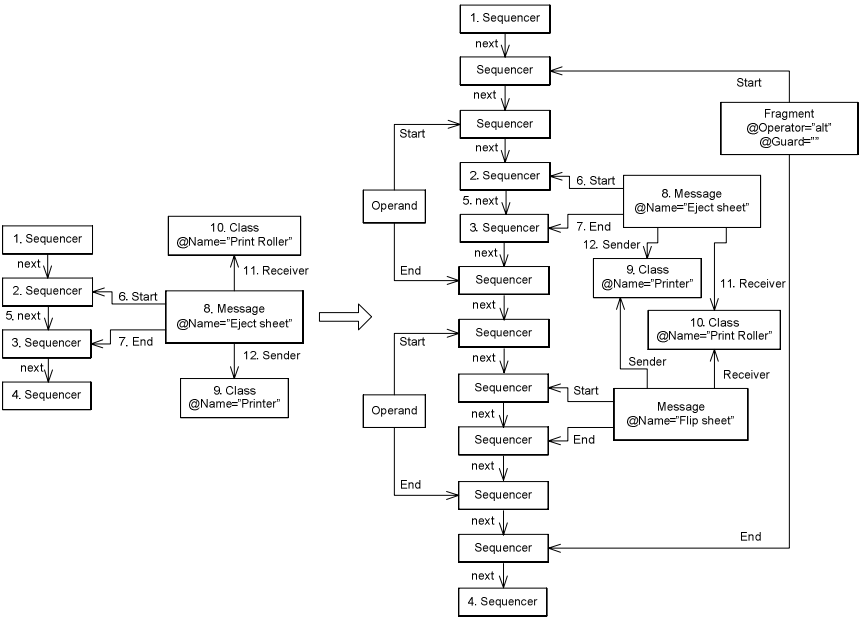

Fig. 29. Graph syntax for rule AddFlipMessage