Graphical Composition of State-Dependent Use Case Behavioral Models

Jon Whittle
Information & Software Engineering
George Mason University
Fairfax, VA 22030
jwhittle@gmu.edu

Ana Moreira
Department of Informatics, FCT
Universidade Nova de Lisboa
2829-516 Caparica, Portugal
amm@di.fct.unl.pt

João Araújo
Department of Informatics, FCT
Universidade Nova de Lisboa
2829-516 Caparica, Portugal
ja@di.fct.unl.pt

Rasheed Rabbi
Information & Software Engineering
George Mason University
Fairfax, VA 22030
rrabbi@gmu.edu

Abstract

Maintaining a clear separation of concerns throughout the software lifecycle has long been a goal of the software community. Concerns that are separated, however, must be composed at some point. This paper presents a technique for keeping state-dependent use cases separate throughout the software modeling process and a method for composing state-dependent use cases. The composition method is based on the graph transformations formalism. This provides a rich yet user-friendly way of composing state dependent use cases that is built on solid foundations. To evaluate our approach, it has been applied to seven student design solutions. Each solution was originally developed using a use case-driven methodology and was reengineered to evaluate whether our technique could have been applied. The findings are that it is *possible to maintain the separation of state-dependent use cases during software design and, furthermore, that expressive model composition methods are necessary to do this in practice.*

1. Introduction

In software engineering, a concern is anything that is of interest to one or more stakeholders, such as a feature, a component or a non-functional requirement. Separation of concerns is the process of isolating key elements in a software system so that these elements can be developed and reasoned about independently. Separating concerns has been recognized as a way of tackling complexity in requirements engineering (e.g., viewpoints [1], aspect-oriented requirements engineering [2]), software architecture (e.g., architecture defined by a set of views [3]), software design (e.g., multi-dimensional separation of concerns MDSOC [4]) and coding (e.g., AspectJ [5], Hyper/J [6]). Recent approaches to separating concerns, such as multi-dimensional separation of concerns and aspect-oriented software development, arose because existing formalisms for developing software artifacts led to a dominant decomposition based on a single concern. This made it difficult to incorporate other concerns.

The result is concern scattering (each concern is implemented in multiple objects) and concern tangling (a single object implements multiple concerns). These problems, of course, are not limited to code but can be found at all stages of software development. A large body of recent work (generally referred to as Aspect Oriented Software Development) has addressed how to support separation of concerns during requirements [2], architecture [7], design [8] and testing. A key problem to be solved by such approaches is: How to compose a set of separated concerns? Concern composition (called *weaving* in aspect-oriented programming) is necessary so that the entire set of concerns can be inspected, analyzed, validated or executed as a whole.

This paper focuses on concern composition during software design. In this paper, we suggest the use of graph transformations [9] to define design compositions. We focus on state-dependent designs given as UML state diagrams. The paper presents a composition language for UML state diagrams based on graph transformations and shows how it can be used to support aspect oriented software development. Although the focus here is on state diagrams, the technique applies to the entire UML. To evaluate our

approach, we apply it to seven student design solutions. These solutions are UML designs that were created as part of a graduate course on software design. They were produced following a traditional use case-based methodology. We re-engineered the solutions to keep separate the state diagrams derived from different use cases and applied our composition technique to compose the state diagrams. This exercise provided evidence that an expressive composition language is necessary for practical UML designs and that our composition language is sufficiently expressive.

The remainder of this paper is structured as follows. Section 2 introduces motivation for our work. Section 3 presents our solution using graph transformations. Section 4 looks at existing design solutions to validate the approach. Section 5 compares related work and is followed by suggestions for future work in Section 6.

2. Motivation

In a recent book [10], Jacobson and Ng argue that use cases are an ideal dimension along which to decompose not only software requirements but also designs and code. In traditional object-oriented design, requirements are decomposed into use cases but implementations are decomposed into objects. This decomposition mismatch is overcome by transforming use cases into objects. Indeed, this transformation is one of the major activities in object-oriented software development. According to Jacobson and Ng, use case decomposition can be maintained throughout design and coding by modeling each use case separately (or more generally, sets of use cases) as a set of objects that form a *use case slice*. These use case slices are then composed at the code level using aspect-oriented programming (AOP) techniques. The result is that the requirements and implementation match more closely and so it is easier to reuse requirements, modify requirements, and maintain traceability links between requirements and code. This idea is, in essence, an application of the more general philosophy called multi-dimensional separation of concerns (MDSOC) [4,11].

Although Jacobson and Ng present a methodology for developing use case slices, they do not fully address how to compose behavior in use case slices *during design*. In fact, their focus is on how to design and compose UML class diagrams and behavioral modeling is, for the most part, ignored. Composition during design is necessary for inspection, analysis and validation of the complete design model.

In this paper, we address how to compose the behavioral parts of use case slices given as UML state diagrams. The problem to be addressed, therefore, is: Given two (or more) state diagrams, each representing the behavior in one use case slice, what is the best way to specify and apply the composition of those designs?

We define the following requirements for a model composition language.

1. **Expressiveness**. A given pair of models can be composed in many different ways and the choice depends on the application. Hence, a composition language must be able to express all possible compositions in as natural a way as possible.
2. **Scalability**. A composition language should scale to large industrial models.
3. **Usability**. A composition language should be as non-invasive as possible, in the sense that practicing software designers should not have to learn a new complex language. A composition language should resemble the design language as closely as possible. Since UML is graphical, this means that its composition language should also be graphical.
4. **Formality**. Although UML does not have a formal semantics, formality is nevertheless a desirable feature. Therefore, a UML composition language should have a formal basis – but one that does not act as a barrier in practice.

A search of the literature on design composition languages reveals that there have generally been two approaches for composing software models. The first is based on aspect-oriented programming. In particular, researchers have taken the syntax of composition mechanisms in AspectJ (pointcuts, joinpoints and advices) and have applied them at the design level [12]. The second is based on Tarr et al's work on multi-dimensional separation of concerns [11]. In this approach, tools provide a default merge algorithm which is predefined and matches elements in the two models. Matching is typically based on the names of the model elements and the default merge algorithm can be modified by textual composition overrides.

Neither of these composition approaches satisfies the four requirements of a model composition language given above. AspectJ-like mechanisms are not very expressive because there is a predetermined set of possible advices that can be used in composition – namely, before, after and around. These mechanisms do not scale to large models because a large model must be broken down before composition if different parts of the model are to be merged using different advices. AspectJ-like mechanisms do tend to be usable, even though they are not graphical, because specifying that a design should be inserted before or after another

is somewhat natural. AspectJ-like mechanisms are typically not formally defined.

On the other hand, MDSOC approaches using a generic merge algorithm are not expressive, scalable or usable. Since a default merge algorithm is used, not all compositions can be expressed easily. It is usually possible to override the default composition but approaches to do this are very low-level and not graphical, leading to problems in scalability and usability.

Our approach is instead based on graph transformations. A graph transformation [9] is a graphical rule that defines modifications on a given graph. Since they have a formal underpinning, they satisfy requirement (4) above. Graph transformations can be used to specify any composition and hence also satisfy (1). Since they are graphical, they at least in part satisfy (3). To address (2) and to further address (1) and (3), the paper presents an analysis of seven student design solutions to see if graph transformations could have been used to specify compositions in practice. We have not yet applied the approach to industrial designs and leave this as future work.

3. A Model Composition Language Based on Graph Transformations

In this section, we present our approach for specifying model composition. We focus in this paper on the behavior of use case slices represented as UML state machines. We present a language for defining compositions that is graphical and formal and closely resembles UML. Although we apply this language to UML state diagrams in this paper, the approach can be adapted to other UML diagrams. We first give an example of a model composition. We then present background on graph transformations. After this, we show how to adapt graph transformations to define a UML state diagram composition language. Finally, we illustrate the expressiveness of our language and compare it to existing composition approaches.

3.1. Example State Diagram Composition

Figure 1 shows two use case slices for a distributed application. The left-hand-side is a use case slice for calling a remote service and consists of a state dependent class ServiceController and a state diagram that defines its behavior. Similarly, the right-hand-side is a use case slice for Handle Network Failure containing the same class ServiceController but with a different set of attributes and a different state diagram. This use case slice describes a limited number of attempts to retry a service after a network failure. Each slice contains the static and dynamic models relevant to a single use case. (A use case slice can, in general, contain more than one use case.) Note, in particular, that each use case slice contains only behavior necessary for that slice. The complete design (in this illustration, comprising only two use cases) is the composition of the two use case slices.

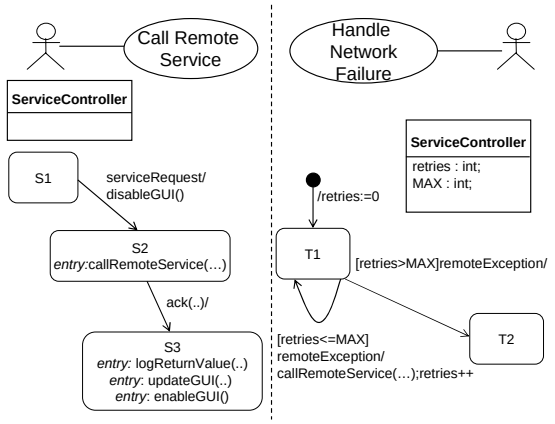

Figure 1: State-Dependent Use Case Slices

During a traditional design process, the two state diagrams for ServiceController would be merged manually and the link between requirement and behavioral design would henceforth be lost. Traceability information could be maintained to keep the link between the states and their use case but this is rarely done in practice because of the complexity of maintaining traceability links as the design is modified.

We propose, instead, that, as in [10], the use case slices be kept as separate concerns throughout development. This makes it easier to modify one use case slice without affecting the other. To inspect or analyze the complete design behavior, the two state diagrams must be composed, preferably automatically. Hence, the use case slices must be accompanied by a composition specification written in an appropriate composition language.

Figure 2 shows the composed state diagram for the two use case slices. This example is deliberately chosen because the composition is non-trivial. The two state diagrams are *interleaved* in a way to satisfy the overall requirements. It is *not* the case, for example, that the state diagram for Handle Network Failure can simply be inserted before or after a single state in the diagram for Call Remote Service. In fact, neither the AspectJ-like or MDSOC approach to composition works very well in this case (see section 4). A more expressive model composition language is necessary.

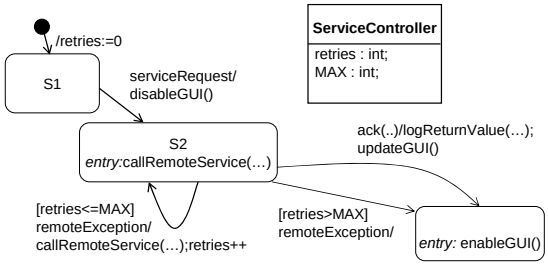

Figure 2: Desired Composition of Use Case Slices

3.2. Graph Transformations

A graph consists of a set of nodes and a set of edges. A graph transformation [9] is a graph rule *r: L* → *R* from a left-hand-side (LHS) graph *L* to a right-hand-side (RHS) graph *R*. The process of applying *r* to a graph *G* involves finding a graph monomorphism, *h*, from *L* to *G* and replacing *h(L)* in *G* with *h(R)*. To avoid "dangling edges" – i.e., edges with a missing source or target node – *h(R)* must be pasted into *G* in such a way that all edges connected to a removed node in *h(L)* are reconnected to a replacement node in *h(R).* An alternative is just to remove all dangling edges.

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 [13]). This work requires that UML models be represented as graphs. The approach is to define node types as the metaclasses in the UML metamodel. For example, a UML class diagram has metaclasses Class, Association, Operation, etc., which become node types. Hence, in this approach, the following are required to define a graph transformation on a UML diagram: a metamodel defining the UML diagram, and a graph rule describing how instances of metaclasses are manipulated.

As an example, Figure 3 shows a fragment of the UML state machine metamodel. Each transition has a source and target state. 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 and transition triggers have names (as represented by a generalization relationship to the abstract class namedElement).

Figure 4 is a 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 [13]. (We defer to [13] for the subtleties of this notation.) Nodes in the graph are given as rectangles. Nodes are attributed and typed so UML class 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*. The LHS in Figure 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 of it) thus moving the transition down in the state hierarchy. Figure 5 shows an example.

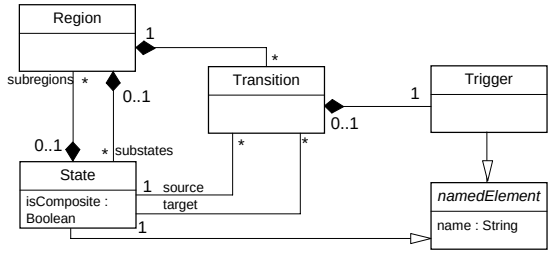

Figure 3: UML State Machine Metamodel

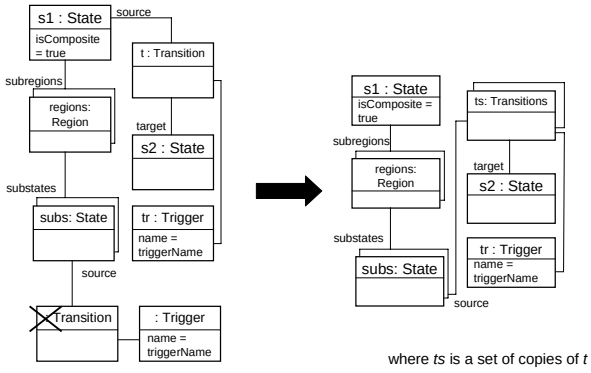

Figure 4: Graph Rule to Move Down Transitions

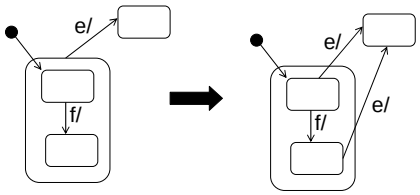

Figure 5: Application of Graph Rule from Figure 4

3.3. A Model Composition Language based on Graph Transformations

The composition of use case slice models can be viewed as an example of a graph transformation if the models can be given as graphs. This can be done by mapping models to instances of their defining metamodel. For example, if Figure 5 is represented as an object diagram showing instances of the State, Transition and Trigger metaclasses, then this object diagram is just a graph. Composition rules for merging use case slice models could be expressed using the notation of Figure 4. In this case, the rules are written over the abstract syntax (i.e., over the metaclasses) of the model rather than the concrete graphical syntax.

For use case slice composition, however, model developers must write the composition rules. Hence, a composition language based on UML metaclasses is impractical because model developers are not generally familiar with the UML metamodel. Furthermore, graph transformations defined over large metamodels are difficult to read and understand [14]. For this reason, we propose *state diagram patterns* as a way to capture the LHS and RHS of a composition rule. State diagram patterns resemble the concrete syntax of UML state diagrams very closely. The concrete syntax is familiar to developers and is therefore more accessible.

For 2 use case slice models, u_1 and u_2, their merge can be defined by a graph rule $c : u_1 \cup u_2 \rightarrow u_{12}$, where u_1, u_2, u_{12} are state diagram patterns and u_{12} represents the merge of u_1 and u_2. Henceforth, c will be referred to as a composition rule. A *state diagram pattern* is an abstract representation of a family of state diagrams and is defined below. Intuitively, a composition rule should capture the two use case slices in as abstract a way as possible. In other words, only model elements relevant to the composition should be included. This keeps the rule general and means that modifications of the use case slices do not usually require modifications of the composition rule.

A state diagram pattern is a state diagram containing pattern variables. Pattern variables are typed over the state machine metaclasses and are marked with multiplicities. Pattern variables are prefixed with a vertical bar '|'. A pattern variable |X has a multiplicity of one. A pattern variable $|X^+$ has a multiplicity of one or more. A 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. Figure 6 gives some examples of state diagram patterns.

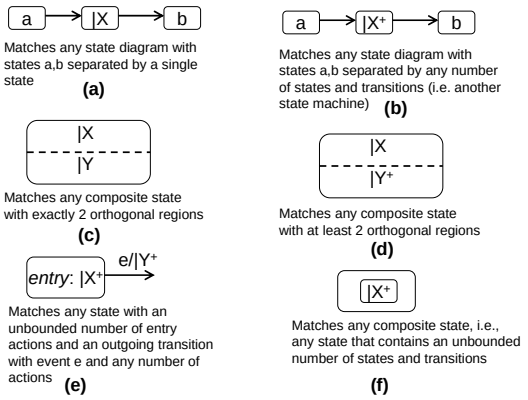

Figure 6: State Diagram Patterns

Patterns based on the notation in Figure 6 are used to describe the LHS of composition rules and also appear on the RHS to show modifications of model elements introduced by the transformation. Figure 6(a), for example, matches any sequence of states starting with a state named *a* and ending with a state named *b*. The variable |X in 6(a) matches any state in between those states. In contrast, the variable $|X^+$ in Figure 6(b) matches any *state diagram* in between *a* and *b*. In a similar way, Figures 6(c) and (d) show how to match against a specific number of regions and an unknown number of regions, respectively. Figure 6(e) is self-explanatory. Figure 6(f) matches a state which contains a pattern $|X^+$ - i.e., there must be at least one substate.

State diagram patterns re-use the concrete syntax of UML state diagrams wherever possible. New notation is introduced only to represent pattern variables and to represent composite states (see Figure 6(f)). The latter is necessary because composite states are given by a meta-attribute in the UML state machine metamodel (*isBoolean* in Figure 4) so it is not possible to distinguish a simple or composite state based purely on the concrete syntax of state. Note also that state diagram patterns need not be valid state diagrams – Figure 6(e), for example, has no target state. State diagrams must be well-typed. The abstract syntax of state diagram patterns is defined by an extension of the

metamodel in Figure 3 and their semantics is given by mapping them to the notation used in Figure 4. Neither is shown here due to lack of space.

More generally, we would like to extend the pattern approach to any modeling language defined by a metamodel. This would involve defining a generic process for starting with a metamodel and producing a pattern language that is close to the concrete syntax. In general, this is difficult because composition rules often need to refer to meta-level concepts that cannot be represented using concrete syntax. Furthermore, metadata cannot be represented in concrete syntax so a composition rule based on concrete syntax would, in some cases, be unable to distinguish between two metaclasses.

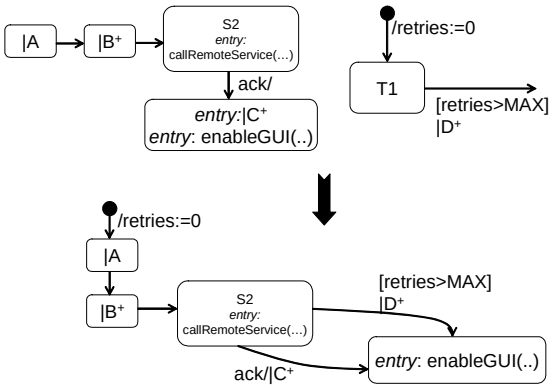

Figure 7: Composition of Use Case Slices

3.3. Model Composition Language Example

This section gives an example composition definition using the model composition language in the previous sections. Recall the two behavioral use case slices in Figure 1. Figure 7 is the model composition definition that will merge these two slices and produce the composed model as given in Figure 2.

The LHS of the graph rule (top half of Figure 7) defines two patterns to match when applying the rule. The first pattern captures a successful service request that starts with *callRemoteService*, is acknowledged, and ends with enabling the GUI. The composition does not depend on any other event/actions that may be present so these are abstracted by pattern variables. The second state diagram pattern defines any situation in which a counter is initialized and a threshold is placed on this counter. Again, any extraneous messages that may occur in a particular application are not included in the pattern. In this way, the pattern is kept as general as possible.

The RHS of the graph rule (bottom half of Figure 7) defines how the two LHS patterns should be merged, again in general terms. When matching the LHS against the state diagrams in Figure 1, the following variable instantiations would occur:

- |A matches state *S1*
- $|B^+$ matches *serviceRequest/disableGUI()*
- $|C^+$ matches *logReturnValue(..)* and *updateGUI(..)*
- $|D^+$ matches *remoteException*

The composition in Figure 7 is non-trivial for two reasons. First, the Handle Network Failure state diagram is split into two parts – one for initializing the retry counter and one for handling exceptions. Each of these parts is inserted into the Call Remote Service state diagram in different places. Secondly, note that the GUI is disabled before the remote service is requested. The GUI is re-enabled (*enableGUI(..)*) both in the case of success and if the maximum number of retries is exceeded. Because of this, state S3 in Figure 1 needs to be split. *logReturnValue()* and *updateGUI()* only occur in the case of a successful remote service call. Hence, in Figure 7, they are placed on their own transition and the entry action *enableGUI(..)* is given its own state.

The next subsection shows that this composition would not have been possible using existing approaches to model composition.

Since our composition language is based on graph transformations, an execution engine for composition can be built using well-known graph transformation execution semantics. We are in the process of building an execution engine for Eclipse EMF models. This will provide the capability to automate compositions of use case slice models.

3.4. Comparison to Existing Approaches

3.4.1. Applying AspectJ at the design level

A number of works have addressed the problem of aspect model composition by applying AspectJ-like advices at the model level (see, for example, [12]). In AspectJ [5], crosscutting behavior can be inserted at well-defined points in the execution of a base Java program. These well-defined points are called join points and the nature of the insertion may be before, after or around a join point. Applying these concepts to UML state machines, one can define either static or dynamic join points. Static join points are syntactic elements of a state machine; dynamic join points are points in the "execution" of a state machine. Since UML state machines have a relatively well understood operational semantics (see, for example, [15]), dynamic join points can be defined easily, e.g., [16]. However, since models are most commonly used for

communication and documentation, and are not necessarily executed, static join points are perhaps more useful in current modeling practices. Static join points for UML state machines include the basic concepts that form state machine abstract syntax, e.g., states, transitions, actions, events. Hence, one may define an AspectJ-like composition of two state machines by defining that one state machine is inserted at a static join point and is placed before, after or around this join point.

Consider how to define such a composition for the two state machines in Figure 1 in order to produce the state machine in Figure 2. Assuming that the base state machine is on the left in Figure 1, one could try to insert the state machine for Handle Network Failure into the state machine for Call Remote Service. Unfortunately, there is no single join point at which the insertion can be done. A first attempt might insert Handle Network Failure after *entry: callRemoteService(…)* in state S2. This does not work, however, because Figure 2 has the *retries* transition before state S1. More importantly, this type of insertion means that if *callRemoteService(…)* results in too many exceptions (i.e., the state machine transitions to state T2), the GUI will not be re-enabled. However, the GUI needs to be enabled (see Figure 2) whether the remote service call ultimately succeeds or not. Using AspectJ's around advice does not work either. Around is used to control the execution of a join point and typically replaces base behavior. In this case, one might wish to control the *entry: callRemoteService* join point so that if the maximum number of tries is exceeded, the actions *logReturnValue(…)* and *updateGUI(…)* are not invoked. But this is not possible unless *logReturnValue(…)* and *updateGUI(…)* are encapsulated inside *callRemoteService* as sub-actions. In other words, a refactoring of the model would be needed first. We conclude therefore that AspectJ-like composition mechanisms can only be used if the state machine models are refactored first. In this example, they could be made to work by first splitting the state machine for Handle Network Failure into two state machines – one involving only the *retries* transition, and one including everything else – and inserting the *retries* transition before S1 and the rest of the state machine around *entry: callRemoteService(…).* In addition, the state machine for Call Remote Service would have to be refactored as described above.

3.4.2. Applying Generic Merge Algorithms

The other major approach to merging models is to apply a generic merge algorithm that may be customized to a particular application. The usual approach is to specify a mapping between the model elements of the models to be merged. If the result of the merge is not the one desired, then the modeler has to modify the result either manually or using a composition directive language that tunes the merge algorithm. In the example of Figure 1, it is hard to imagine how such an approach could be practical. If the merge algorithm is based on matching state names, one could attempt to merge with the mapping T1=S2. However, the merge result would then have the transition *retries* in the wrong position and, as in the previous subsection, *enableGUI(..)* will not be executed when *retries>MAX*. Hence, manual modifications would have to be made to the merge result to achieve the desired state machine in Figure 2. Although these manual modifications could be specified in a composition directive language so that they could be applied as part of the merge algorithm, composition directive languages such as those in [17,11] are not graphical and do not resemble the UML state machine language. Furthermore, they provide a very low level way of specifying such details and hence, the approach does not scale easily.

4. Evaluation

The previous section showed that a graph transformation approach is more expressive than composition based on either AspectJ or MDSOC. However, the question remains whether the additional expressiveness is actually required in practice. Could it be, for instance, that the example presented in this paper is merely contrived to show the advantages of graph transformations? To answer this question, we undertook an investigation of existing design solutions to see which compositions occur in practice.

Our experiments attempted to answer the following question. In practical examples, are model composition mechanisms based on AspectJ-like or MDSOC-like approaches expressive enough? Moreover, is our approach based on graph transformations expressive enough? Hence, we attempted to address the first requirement of model composition identified in Section 2. The methodology was to examine existing UML designs, to refactor those designs to reflect the use case slice technique of Jacobson and Ng, and then to investigate the level of expressiveness required to compose state diagrams from different use case slices. Because of availability of the models, we chose to study solutions to a term project conducted as graded assignments for a graduate course in software design. This study was conducted in the semester following the class and so did not affect the grading in any way.

We studied seven team project designs, each expressed in UML consisting of use cases, class diagrams, interaction diagrams and state diagrams. Only the use cases and state diagrams were relevant to the study. 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 resulting designs, however, were quite different because teams worked independently.

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. To give a sense of the size, there were 10 use case slices and an average of 12 state diagrams for each solution.

Based on an analysis of the compositions of state-dependent use case slices, we identified four categories of composition. The next subsections describe these and give examples in each case.

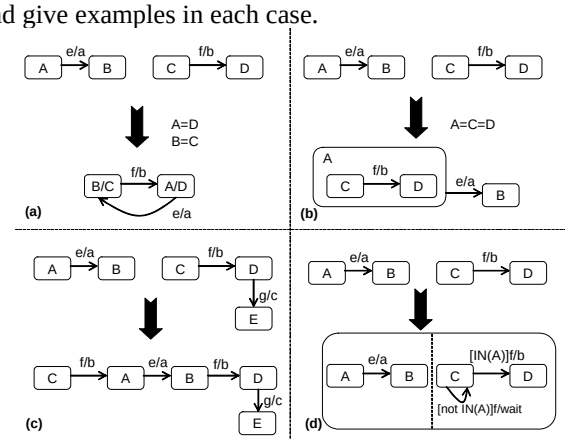

Figure 8: Composition Categories

4.1. One-to-one state matching

The first category contains 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 8(a) gives an example. In the student solutions, this case occurred typically when two state diagrams defined sequences that were joined together into a loop.

4.2. Many-to-many state matching

This category is an extension of the previous one whereby states in the two LHS 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 Figure 8(b)). This type of composition is not typically employed by MDSOC but has been investigated in previous work by authors of this paper [18]. Figure 8(b) gives an example.

4.3. State diagram refactoring

In this category, one or more of the LHS 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 of composition cannot be handled by state matching because state matching cannot refactor a state diagram. Figure 8(c) illustrates.

4.4. 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 use case slice only 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 8(d) gives an example.

4.5. Discussion

Based on the student design solutions, we found that all four categories of composition were common. The breakdown for the four categories was as follows: 13%, 39%, 46%, 2%. Only the graph transformation approach is expressive enough to support all four categories. The MDSOC approach supports only category 4.1 although it can be easily extended to support 4.2 (as was done in [18]). The AspectJ-approach does not support either category 4.1 or 4.2 because both categories allow complex interleavings that cannot be expressed using just before/after advices. Some compositions in 4.3 could be supported by AspectJ if the state diagram to be decomposed is first refactored into multiple state diagrams. Each state diagram fragment can then be inserted at a different place. However, we view this as a non-optimal approach to composition because it involves

representing fragments of a state diagram separately which leads to problems in reusability and readability.

Graph transformations support 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.

5. Related Work

The most common approaches to merging behavioral aspects have already been discussed in Section 3.4. Another approach is that of composition filters [24]. Composition filters are an approach to defining crosscutting concerns on top of OO programs. Filters intercept messages directed towards or away from objects and may manipulate those messages before dispatching or redirecting them. The composition filters approach could also be applied to UML models. Composition filters deliberately keep the crosscutting code and the base code separate – they are not explicitly composed. Explicit composition appears to be desirable for modeling because models are communication tools and concern separation approaches that do not show the composed models are likely to reduce readability. Our composition language applies explicit composition but the trade-off is that the compositions may be harder to express. Further investigation is required to examine these trade-offs.

A number of works have proposed mechanisms for aspect-oriented modeling in UML. The UMLAUT framework [22] is a tool to compose UML models where the compositions are defined over an abstract syntax tree. UMLAUT uses a textual transformation language over metaclasses that is not easily accessible for software modelers. Some approaches [8,11] use UML templates to represent aspect models. These have mainly focused on UML class diagrams, however. Stein et al [25] have introduced joinpoint designation diagrams, a graphical way of identifying joinpoints for models. This approach is somewhat more complicated than ours and it is not clear if these diagrams are understandable by model developers. Furthermore, Stein does not consider how to compose models based on these joinpoint specifications. [26] composes state machines by using orthogonal regions. However, this makes it difficult to visualize the complete behavior and so may not support model inspection. The C-SAW tool [27] is a model transformation engine in which aspects can be defined using a textual language.

Graph transformations have a noble history in software engineering. Their use has been suggested for viewpoint integration [19] in requirements engineering, software refactoring [13], and generative programming programming [20]. Graph transformations influenced the MOF 2.0 Query/Views/Transformations (QVT) effort [21]. However, to the authors' knowledge, there has been no in-depth research on composing aspects using graph transformations. In our approach, the graph compositions are defined over UML models and resemble the equivalent UML models very closely. This is in contrast to existing approaches that map models based on the metamodel.

There has been much work on defining composition operators for formal specification languages in a way that preserves semantics. Recent work in this direction, for example, is presented in [28]. Our aim is somewhat different. We aim to provide composition languages that are accessible to working software modelers.

6. Conclusion and Further Work

This paper serves two purposes. First, it presents a novel way of adapting graph transformations to define model composition for aspect-oriented software development (AOSD). Secondly, it acts as one of the first experimental validations of the ideas put forth by Jacobson and Ng [10] of keeping use cases independent throughout the development lifecycle.

The use of graph transformations in model composition is beneficial in the following ways. As shown by the study in this paper, graph transformations are more expressive than existing model composition techniques in the AOSD field. Furthermore, this richer expressiveness *is* required in practice – even relatively small design solutions would require it, for example.

There is a question of the scalability of model composition in general, and the graph transformation approach in particular. We found that even for student design solutions, a graph transformation definition can sometimes become complex and it is unlikely that a working software engineer not trained in formal methods would be able to specify the transformations. On the other hand, MDSOC and AspectJ-like approaches are sometimes (but not always) more intuitive. We feel that, ultimately, a combination of the three approaches would be ideal so that the simpler approaches could be used when the additional expressiveness is not necessary. The categorization of composition types provided in Section 4 can be used as guidelines towards identifying which composition approach is more suitable for a given composition. Note also that we expect there to be more user-friendly ways of defining graph transformations than were given in this paper. Having graph transformations as an underlying formalism is useful for formal analysis, but,

but, in the future, we will investigate how to define these transformations in a more intuitive way.

Currently, no execution engine exists for the model composition language defined in this paper. We are in the process of building such an engine as an Eclipse plug-in. Existing graph transformation tools such as AGG [23] will be used where appropriate. Once our tool is built, users will be able to define EMF UML2.0 compliant use case slices and execute compositions based on definitions in our model composition language. Furthermore, we plan to investigate whether graph transformations can offer any solutions to the aspect interaction problem – whereby application of multiple compositions may have unexpected effects. We plan to use existing graph transformation analysis techniques (such as critical pair analysis) to help in identifying conflicts.

The composition language in this paper is syntax-based. This can make the technique brittle in the presence of minor syntactic changes. Future work will investigate if semantic knowledge of models can be exploited in composition.

7. References

[1] B. Nuseibeh, J. Kramer, A. Finkelstein, "ViewPoints: meaningful relationships are difficult!" ICSE 2003, 676-683.

[2] A. Moreira, A. Rashid, J. Araújo, "A. Multi-Dimensional Separation of Concerns in Requirements Engineering," The 13th International Conference on Requirements Engineering (RE'05), IEEE Computer Society, France, 2005.

[3] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, J. Stafford, "Documenting Software Architectures: Views and Beyond," Addison Wesley, 2002.

[4] P.L. Tarr, H. Ossher, W.H. Harrison, S.M. Sutton Jr., "*N* Degrees of Separation: Multi-Dimensional Separation of Concerns." ICSE 1999: pps 107-119.

[5] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W.G. Griswold, "An overview of AspectJ." ECOOP Budapest, Hungary, June 2001, pp. 327–353.

[6] H. Ossher, P.L. Tarr, "Hyper/J: multi-dimensional separation of concerns for Java." ICSE 2000, pps. 734-737.

[7] B. Tekinerdogan, "ASAAM: Aspectual Software Architecture Analysis Method," Working IEEE/IFIP Conference on Software Architecture, Oslo, June, 2004.

[8] R. France, I. Ray, G. Georg and S. Ghosh, "An Aspect-Oriented Approach to Design Modeling", IEE Proceedings Software, Vol 151(4), pp. 174-186, August 2004.

[9] G. Rozenberg, editor. "Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations." World Scientific, 1997.

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

[11] S. Clarke and E. Baniassad, "Aspect-Oriented Analysis and Design: The Theme Approach," Addison Wesley, 2005.

[12] D. Stein, S. Hanenberg and R. Unland, "A UML-based Aspect-oriented Design Notation for AspectJ." 1st International Conference on Aspect-oriented Software Development, pages 106–112. ACM Press, 2002.

[13] S. Markovic and T. Baar, "Refactoring OCL annotated UML class diagrams", *In Model Driven Engineering Languages and Systems*, 8th International Conference, MoDELS 2005, Montego Bay, Jamaica, 2005, 280-294.

[14] Baar, T. and Whittle, J., "On the Usage of Concrete Syntax in Model Transformation Rules," Sixth International Andrei Ershov Memorial Conference PERSPECTIVES OF SYSTEM INFORMATICS, 2006, Springer LNCS.

[15] M. von der Beeck. "A structured operational semantics for UML-statecharts." *Software and System Modeling*, 1(2):130.141, 2002.

[16] S. Nakajima and T. Tamai, "Aspect-Oriented Software Design with a Variant of UML/STD." 2006 Workshop on Scenarios and State Machines, at ICSE 2006.

[17] G. Georg and R. France. "UML Aspect Specification using Role Models."Proceedings of 8th International Conference on Object Oriented Information Systems, Springer, LNCS Vol. 2425, 186-191, 2002.

[18] J.Araújo, J. Whittle and D-Kim, "Modeling and Composing Scenario-Based Requirements with Aspects", 12th IEEE International Requirements Engineering Conference (RE), Japan, IEEE CS Press, September 2004.

[19] M. Goedicke, B. Enders, T. Meyer, G. Taentzer, "ViewPoint-Oriented Software Development: Tool Support for Integrating Multiple Perspectives by Distributed Graph Transformation." TACAS 2000: 43-47

[20] S. Sendall, "Combining Generative and Graph Transformation Techniques for Model Transformation: An Effective Alliance?" OOPSLA '03 Workshop "Generative techniques in the context of MDA", 2003.

[21] OMG. "Meta Object facility (MOF) 2.0 Query/ View/Transformation Specification." OMG Document ptc/05-11-01, Nov 2005.

[22] UMLAUT: Unified Modeling Language All pUrpose Transformer, http://www.irisa.fr/UMLAUT/

[23] J. de Lara, G. Taentzer, "Automated Model Transformation and Its Validation Using AToM 3 and AGG." Diagrams 2004, pps 182-198.

[24] L. Bergmans, M. Aksit, "Composing crosscutting concerns using composition filters." Commun. ACM 44(10): 51-57 (2001)

[25] D. Stein, S. Hanenberg, R.Unland, "Join Point Designation Diagrams: A Graphical Representation of Join Point Selections," International Journal of Software Engineering and Knowledge Engineering, Vol. 16(3), 2006.

[26] M. Mahoney, A. Bader, T. Elrad, O. Aldawud, "Using Aspects to Abstract and Modularize Statecharts," The 5th Aspect-Oriented Modeling Workshop in Conjunction with UML 2004 Lisbon, Portugal, October 2004

[27] J. Zhang, Y. Lin, and J. Gray, "Generic and Domain-Specific Model Refactoring using a Model Transformation Engine," in *Model-driven Software Development*, Springer, 2005, Chapter 9, pps. 199-218.

[28] G. Brunet, M. Chechik, S. Uchitel, "Properties of Behavioural Model Merging." FM 2006, 98-114.