Implementing Aspect-Oriented Modeling and Weaving in AToM^3[2]

Wisam Al Abed[1]

School of Computer Science, Mcgill University,
Montreal, QC, Canada,
wisam.alabed@mail.mcgill.ca

Abstract. This paper discusses an approach by Kienzle et al.[1] to modeling aspects within the context of transactional frameworks. It touches upon the aspect-oriented modeling (AOM) and feature modeling techniques used. The AOM approach allows the definitions of stand-alone reusable aspect models, and supports the modeling of structure(using UML class diagrams) and behavior (using UML sequence diagrams). The feature modeling approach allows the handling of variabilities in the product line built with several aspect models. We discuss the notion of inter-aspect weaving, which allows an aspect to reuse functionality modeled by other aspects. The resulting aspect model can be woven with an application model in order to create a structural and behavioral model of the application augmented with the functionality modeled by the aspect. Furthermore, we take this a step further by attempting to implement a subset of this approach in a meta-modeling and graph rewriting tool AToM^3[2]. Our goal is to visually represent such models and then use graph rewriting to perform the weaving.

1 Introduction

In order to use aspect-oriented modeling in a viable manner, there is a need to introduce a notion of weaving at the modeling level. The motivation for this, as the paper [1] points out, is to allow for simpler and potentially more intuitive ways of validation, simulation and code generation of the system that is being modeled. The paper[1] presents an approach where any concern or functionality that is reusable is modeled as an aspect: the functionality defined within the aspect cuts across all the applications in which the functionality is reused. Hence, the approach presented allows the modeler to specify the structure and behavior of a concern in a reusable aspect model. Furthermore, the paper[1] applies this to the transactions domain for the sake of having a non-trivial and interesting example to work with. In this paper, We shall touch a little bit upon this approach by examining some of the aspects from the Aspect Optima transactions Framework. We shall then discuss how we implemented this in a meta-modeling tool AToM^3[2]. The paper is structured as follows: Section 2 introduces the background for the Aspect-Oriented approach as presented in the original paper[1]

but restricted to looking at two simple concerns in the transactions framework. Section 3 will introduce AToM^3 and how we implemented the meta-model for aspect modeling and how we used graph rewriting to perform the weaving. Section 4 will discuss some of the issues and limitations to the implementation discussed in section 3. Section 5 will discuss some potential future work and we will conclude and summarize in section 6.

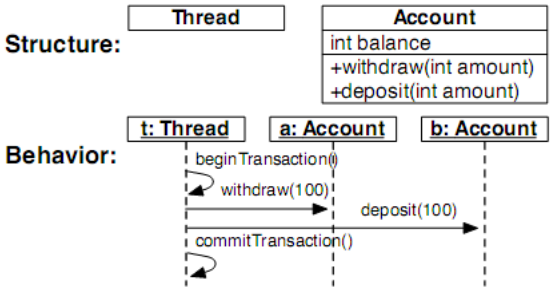

Fig. 1. A simple Application Model[1]

2 Aspect-Oriented Modeling Background

This section introduces the reusable aspect model approach presented by Kienzle et al.[1] that our implementation is based upon.

2.1 The Motivation

Fig.1 defines a simple application model of a bank, where a Thread instance t transfers 100 from account a to account b by calling a's withdraw method followed by b's deposit method. The transfer is enclosed within beginTransaction and commitTransaction invocations. We wish to apply some transactional properties to this application. Fig.2 defines a transaction product line proposed by Kienzle et al.[1]. Each feature can be thought of as a separate concern for a particular transaction that have dependencies amongst one another as portrayed in the feature diagram. To make this more concrete we can, for example, choose particular features in order to create a certain configuration. Fig.3 shows this for a flat transaction configuration. Suppose we wish to have the concerns shown in Fig.3 be applied to our simple bank application. One possibility is that we could,

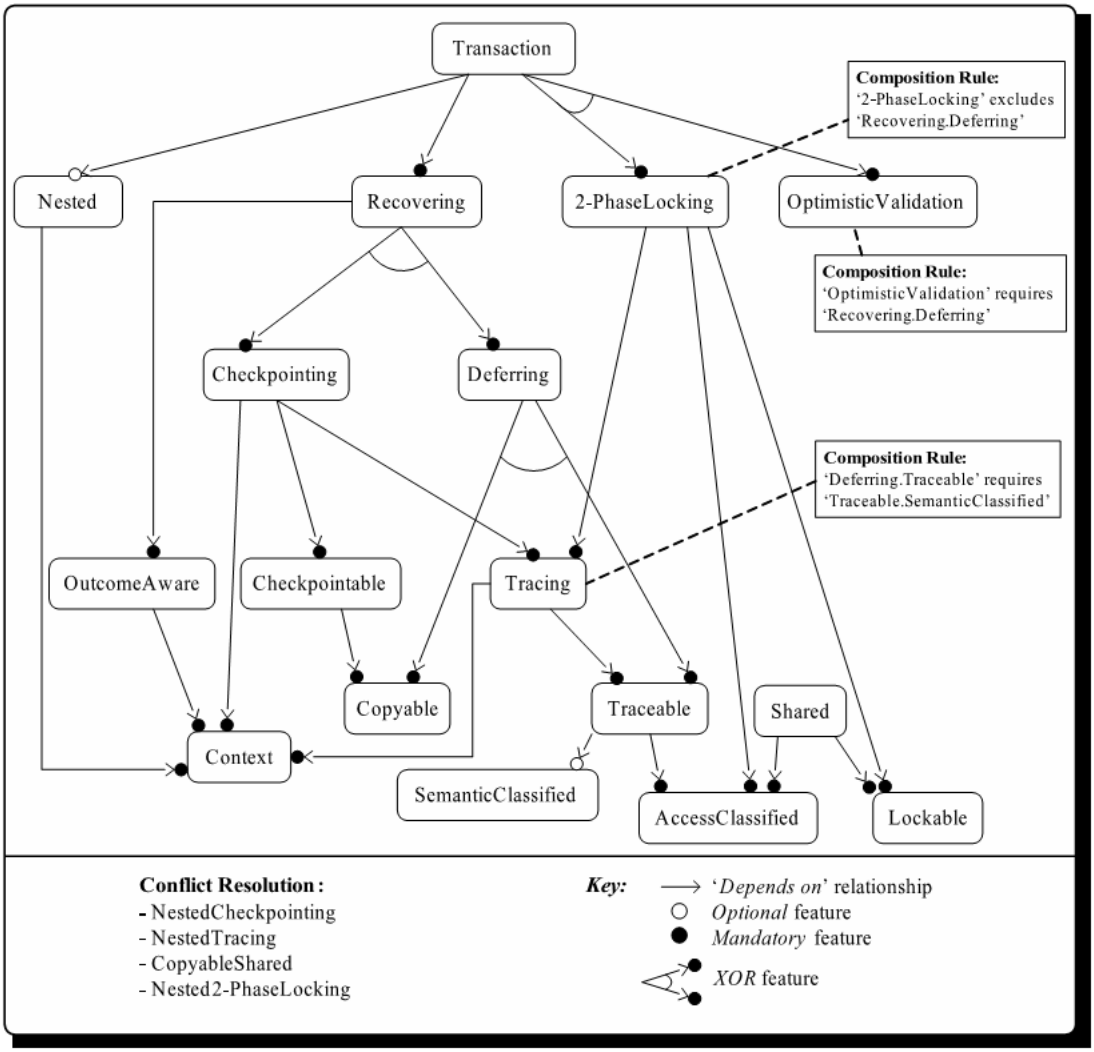

Fig. 2. The product line of transactions[1]

by hand, construct the structure and behavior of this application to conform to a flat transaction; this would involve large amounts of effort and brings forth an increased risk in introducing errors. This approach is also not very modular, hence limiting reusability. Fig.4 and Fig.5 is an example of the structure and behavior after applying the flat transaction to our bank application. It is clear that this is quite complex. As the application increases in complexity one can imagine that this approach will explode exponentially. There must be a better way for us

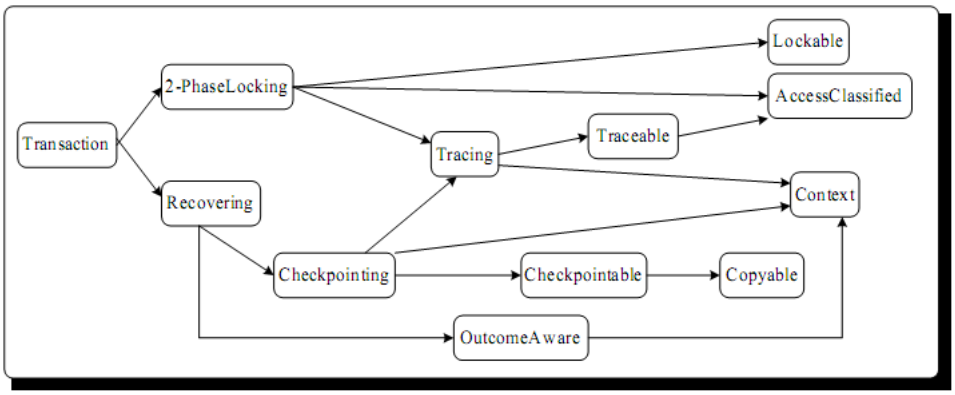

Fig. 3. Aspect Dependency chain for a Flat Transaction[1]

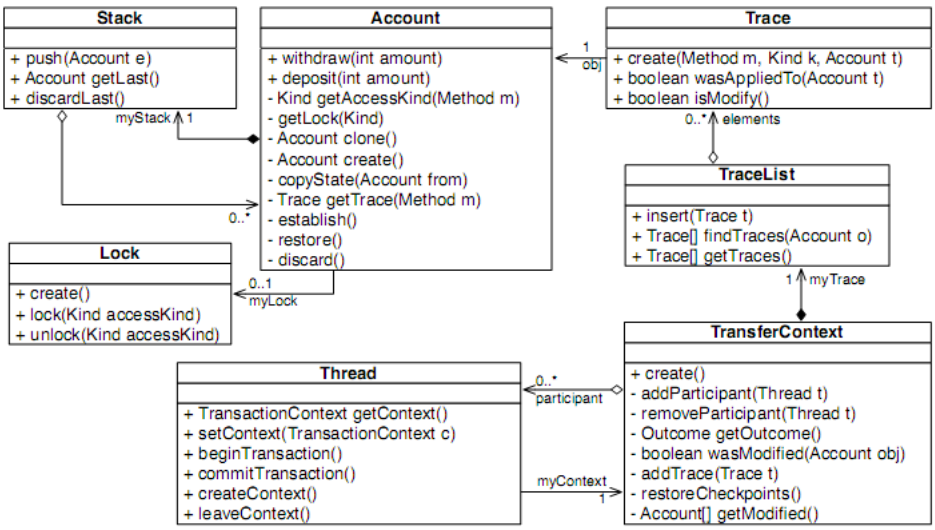

Fig. 4. Structure of Woven Application Model[1]

to modularize this by potentially breaking the construction into smaller pieces and then defining rules for how to combine things together in order to come up with the final weaved model. This is what shall be discussed in the next section regarding the aspect-oriented approach presented in the paper[1] to tackle this issue.

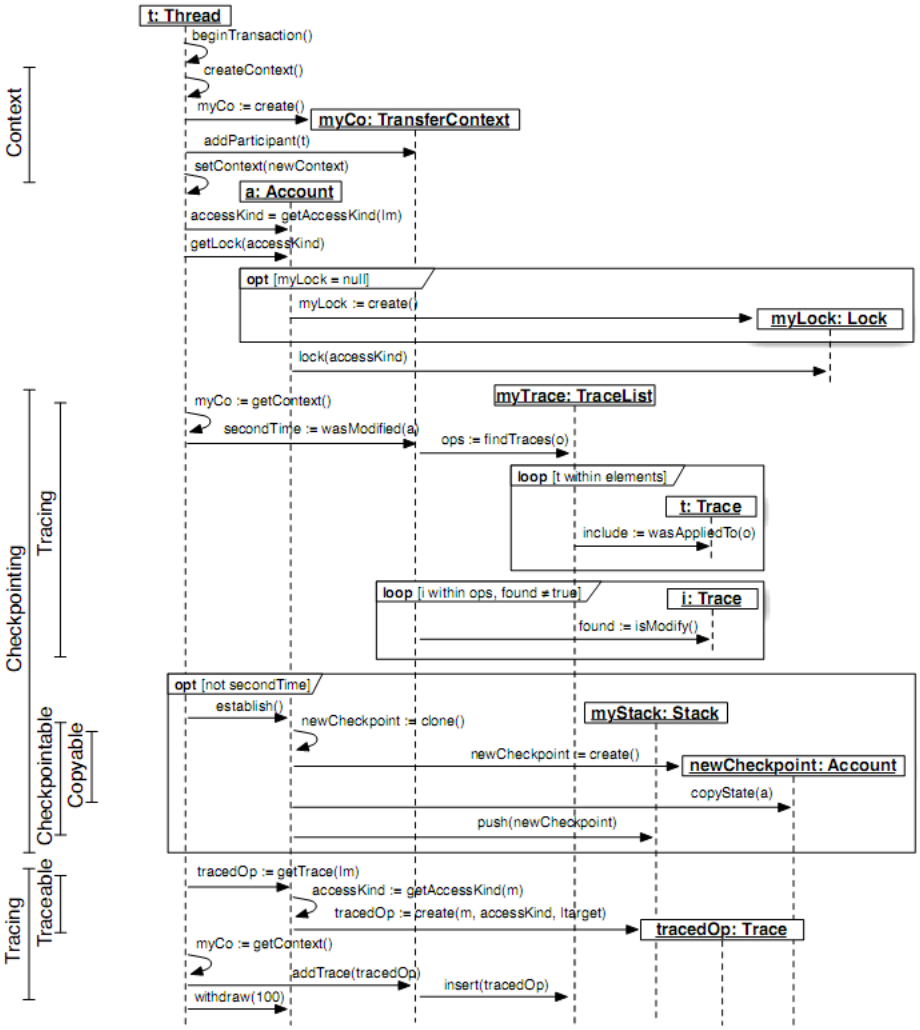

Fig. 5. Behavior of Woven Application Model[1]

2.2 The Aspect Oriented Approach

The paper[1] presents an aspect-oriented approach that allows one to define the structure and behavior of a concern in a reusable aspect model. Then using feature modeling or product line concepts it combines these two ideas to allow a modeler to choose the concerns he desires from the product line resulting in a certain configuration. This is doable since the aspects are reusable and composable as presented in the paper and then these inter-dependent aspects will be weaved together resulting in a final product. So we have a way of creating aspects that are reusable. These aspects can depend on one another creating potential dependency trees. We shall now show two examples of aspects, taken from the Aspect Optima Framework, that outline the modeling notation presented by Kienzle et al.[1].

An Example: Traceable and AccessClassified

An aspect is defined within a UML package identified with the keyword aspect. The aspect structure is defined in a compartment with the keyword structure, followed by behavior compartments for each functionality that the aspect provides that requires message exchanges among objects. If no message exchanges between objects are necessary, then a simple method declaration in the structure compartment suffices. Each behavior is split into two parts with the keywords Pointcut and Advice.

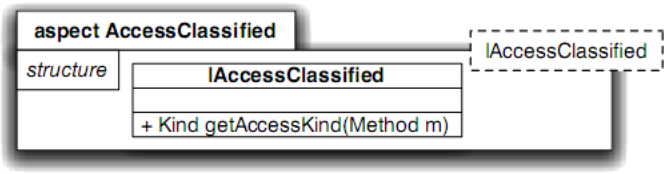

Fig. 6. The *AccessClassified* Aspect[1]

Fig.6 shows the AccessClassified aspect. It has no behavior only structure and simply specifies a class with one method. The paper proposes to use UML template parameters in order to identify within an aspect model which modeling elements are generic. Reusing an aspect model involves binding the aspect models template parameters to target model-specific elements. In the example in Fig. 6, the class |AccessClassified is a template parameter; when wanting to bind this we can declare instantiation directives. For example the following: $AccessClassified.|AccessClassified \rightarrow Transactional*$ specifies that all classes that have names starting with the string Transactional... are extended with the functionality defined in AccessClassified. This is important since once

this is declared successfully we get a context-specific aspect model that can be composed with a target model.

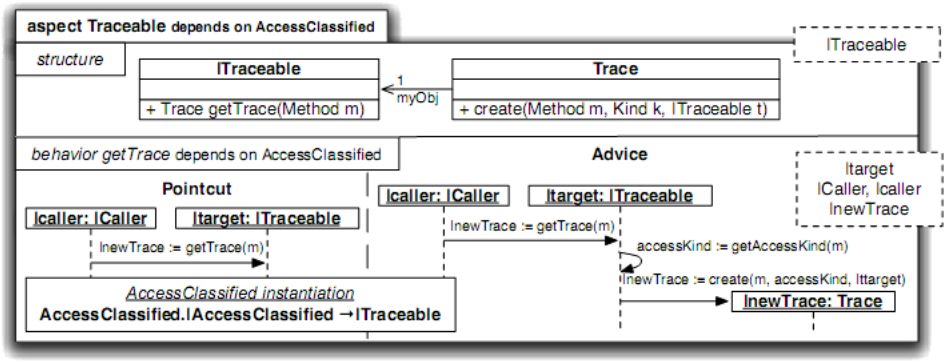

Fig. 7. The *Traceable* Aspect[1]

Fig.7 shows the Traceable aspect which depends on AccessClassified. It contains structure and behavior as well as an instantiation directive that allows us to reuse the functionality in AccessClassified. The instantiation directive within getTrace, $AccessClassified.|AccessClassified \rightarrow |Traceable$, declares that all Traceable objects have to be AccessClassified as well. This is an example of how dependency amongst aspects is handled in the approach presented by Kienzle et al.[1]. This is also important in demonstrating the reusability of the aspects using this approach. Now that we have demonstrated how these aspects get modeled we shall examine how they are weaved. Traceable depends on AccessClassified and we have seen how the instantiation directives define this. So before Traceable can be woven into an application model, AccessClassified must first be woven into Traceable creating an *independent model*. This weaving is like any other weaving. After binding the template parameter of AccessClassified according to the instantiation directive shown above, we attempt to match the pointCut of AccessClassified against the advice of Traceable. Then, any occurrences of the pointcut within the advice of Traceable are composed with the advice of AccessClassified. Of course in this case this is trivial since AccessClassified does not define any behavior, but this technique is a general technique that can be applied to any two aspects that are being woven into one another.

3 The Implementation in AToM^3

AToM^3[2] is a tool for multi-paradigm modeling developed at the Modeling, Simulation and Design Lab (MSDL) in the School of Computer Science of

McGill University. This tool provides two main tasks: meta-modeling and model-transformation. In this paper we will show how we used this tool to implement the aspect oriented modeling approach presented by Kienzle et al.[1]. We took a simple example, a modified version of the bank application seen in Fig.1, and picked two aspects to apply to it Traceable and AccessClassified. We shall show how we modeled these aspects in AToM^3[2] and then how we used the graph rewriting engine to perform the weaving.

3.1 The Meta-Model

First of all in order to model these aspects(Fig.6 and Fig.7) and the base

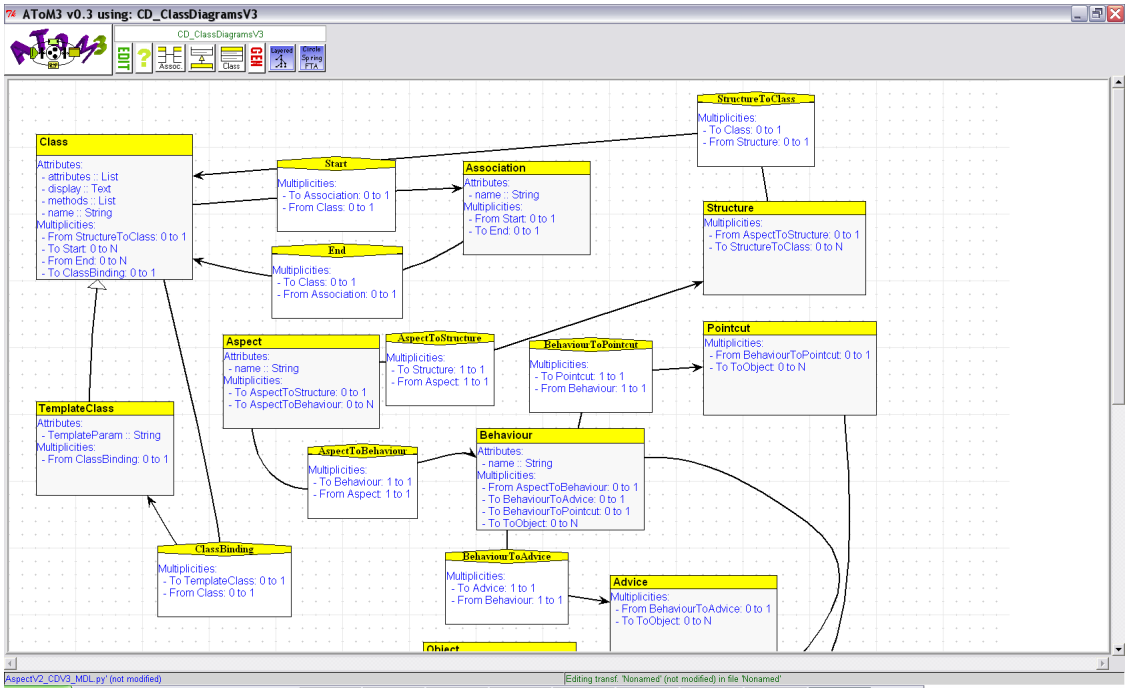

Fig. 8. The First part of the meta-model for aspects in AToM^3

application(Fig.1) in AToM^3[2] we will first need to define a meta-model that will allow us to build such models. We decided to use the class diagrams formalism provided by AToM^3[2] to build our meta-model. Fig.8 and Fig.9 shows the full meta-model as developed using AToM^3[2]. This meta-model has 4 key parts: first it allows for building of class diagrams; second it allows for modeling of sequence diagrams; third it allows for associating these to their respective aspect packages; finally the meta-model allows for binding between aspects as a way of visually

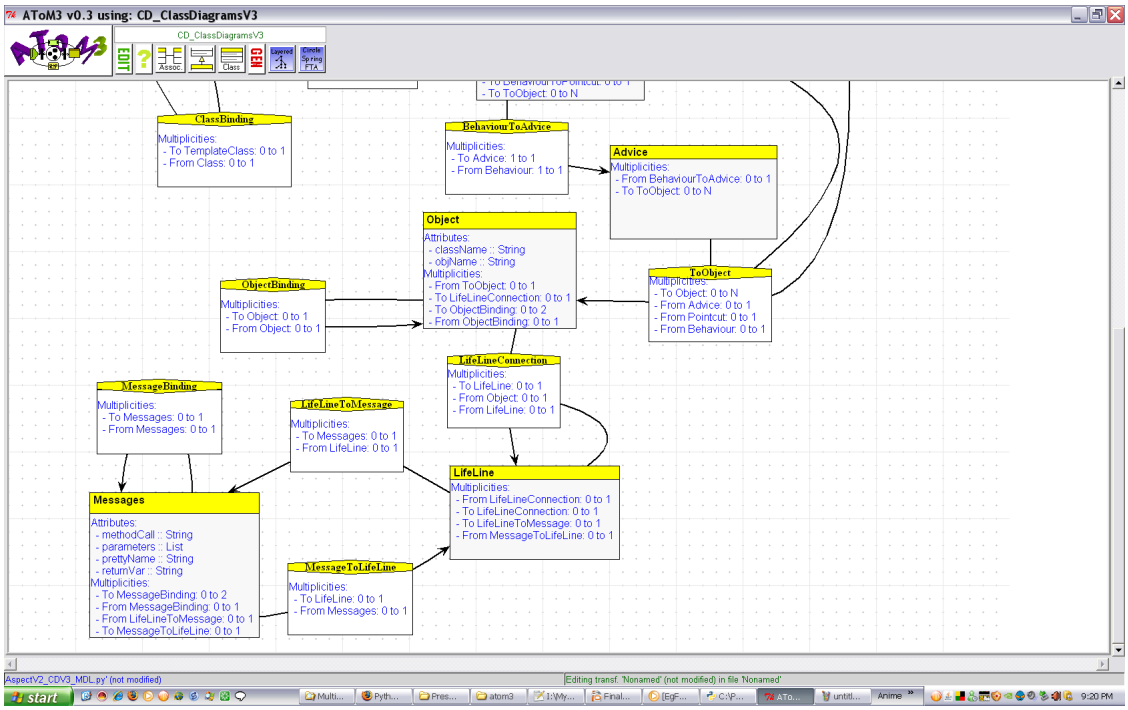

Fig. 9. The Second part of the meta-model for aspects in AToM^3

representing the instantiation directives that Kienzle et al.[1] introduced in their paper. The important part to note is that the template parameters were modeled via template classes and the binding was defined as an association between a template class and another template class or between a class and a template class. Binding is also defined at the sequence diagram level between objects, again in order to visually represent the instantiation directives that Kienzle et al.[1] introduced in their paper. The reason behind this is that the focus of this paper is using graph rewriting to perform the weaving and not the algorithms that perform element matching for class and sequence diagrams.

With this meta-model implemented we can now start building aspect models using this tool. Fig.10 shows how the bank application will look like modeled in the tool. This is a modified version of the original application presented in the paper and this was done this way in order to validate the weaving. The original was meant to be weaved with a transactions aspect but since we have not reached that far in terms of our implementation we needed to simplify the base application. Also take note that this is a special type of aspect that does not contain a notion of Pointcut and Advice rather just one single overall behavior. Fig.11 shows the aspect AccessClassified, you can see that the template parameter is defined using the template class. Fig.12 shows the aspect Traceable, you

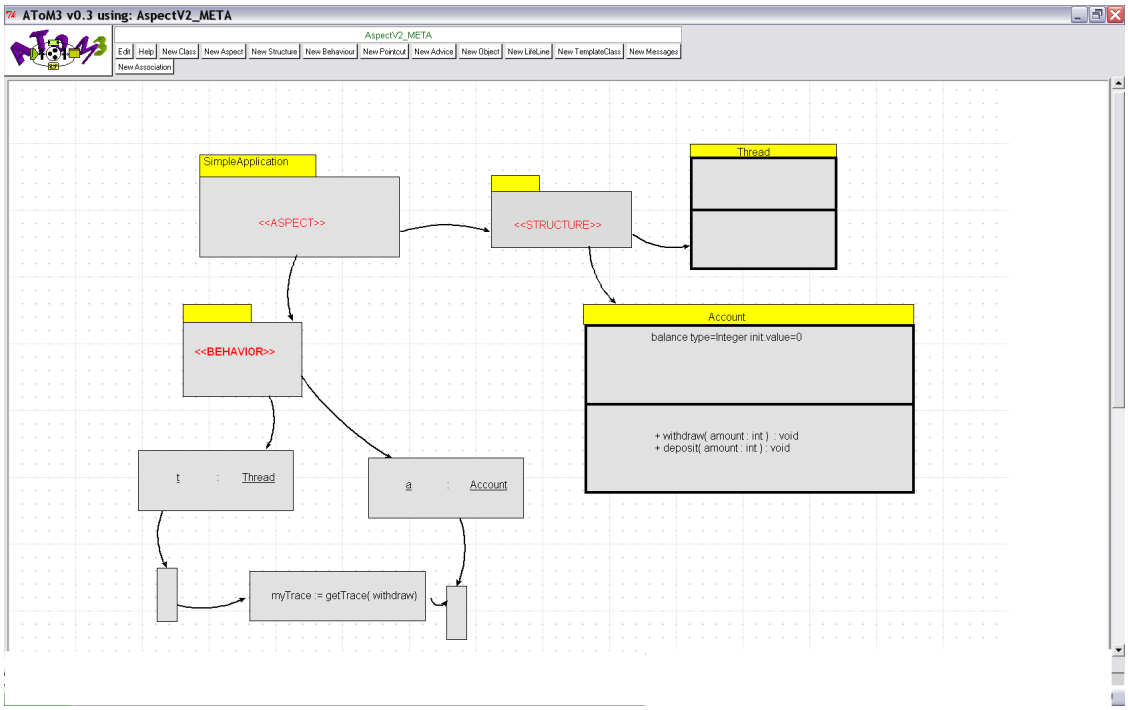

Fig. 10. The Base Application as modeled in AToM^3

can see how the behavior getTrace is split into a Pointcut and Advice just as Kienzle et al.[1] introduced in their paper. Overall, despite the fact that models produced by the tool are not visually the same as the models Kienzle et al.[1] developed, they are the same in that they convey the same key concepts and maintain the same functionality and purpose.

Now that we have built these models we can move onto binding them together in order to set up our example to perform the weaving. Fig.13 shows how the binding looks like from the class diagram perspective. Since we know from the paper[1] that Traceable depends on Accessclassified we must emulate the instantiation directive $AccessClassified.|AccessClassified \rightarrow |Traceable$. We do this by binding the template class |Traceable to the template class |AccessClassified, This will later allow our weaver to detect such a pattern and perform the necessary merging. Analogously we do the same to bind our base application to the aspect Traceable in order to emulate the instantiation directive $Traceable.|Traceable \rightarrow Account$. We create a binding between the Account class and the template class |Traceable. Finally, since we also need to perform binding at the behavioral level we must emulate the following directives: $getTrace.|caller \rightarrow t$, $getTrace.|Caller \rightarrow Thread$, $getTrace.|target \rightarrow a$, and $getTrace.|Traceable \rightarrow Account$. Again, similar to how binding was done

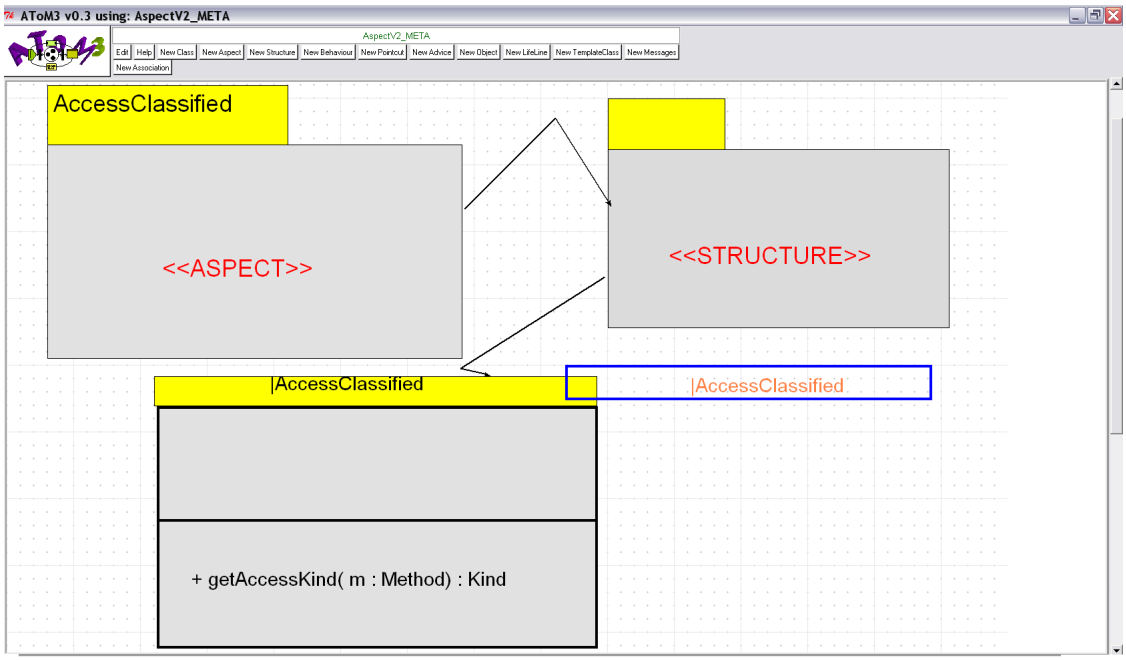

Fig. 11. AccessClassified Aspect as modeled in AToM^3

between classes we do this between objects as seen in Fig.14. The one thing to note about the binding is that it not only binds with objects in the Pointcut of the aspect Traceable but also the Advice. The reason behind this, again, is to forgo any need to perform element matching of Pointcuts and focus on the graph rewriting. Also we added the option of binding between messages this is because there are cases where certain variables need to be bound even though they are not represented as objects in the sequence diagram since they do not participate in any message communication. In this case this applies to the variable |newTrace which is where the return value of the call to getTrace gets stored. Again this was done to facilitate the graph rewriting and make it easier to perform the weaving. Now that our example has been fully modeled we can move on to using graph rewriting to perform the weaving.

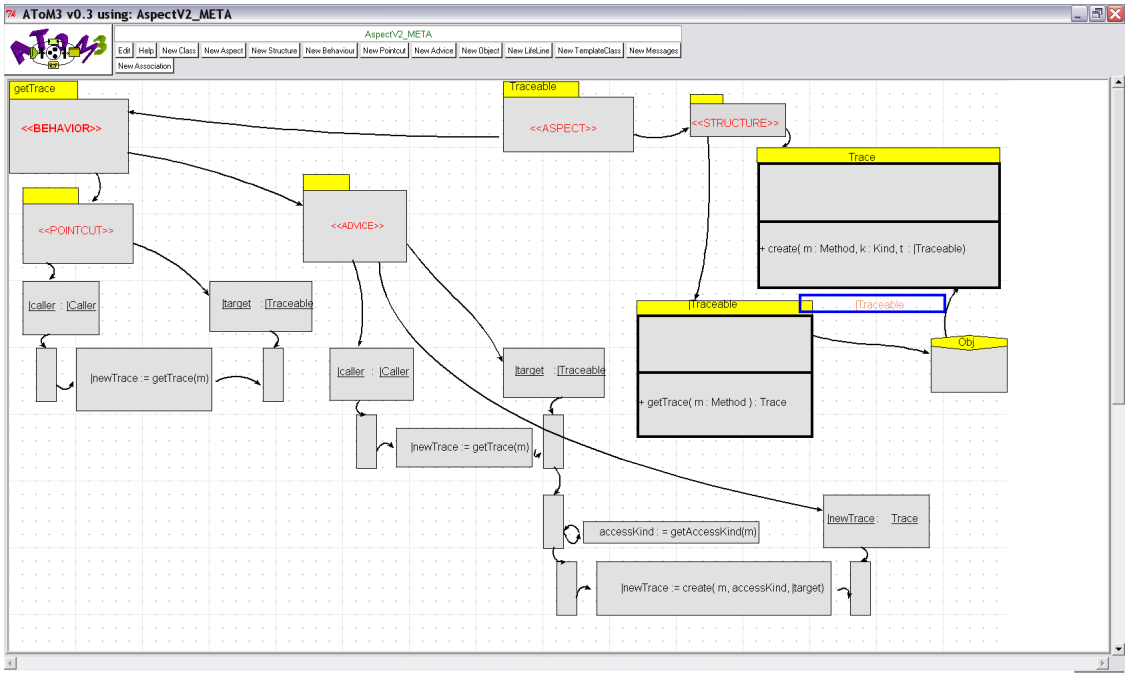

Fig. 12. AccessClassified Aspect as modeled in AToM^3

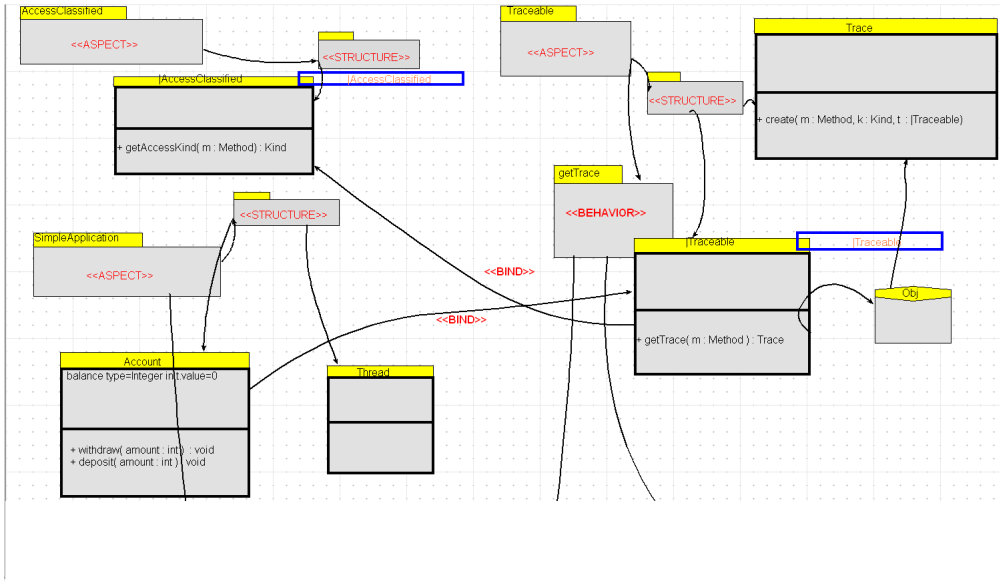

Fig. 13. First part of the binding amongst the aspects modeled in AToM^3

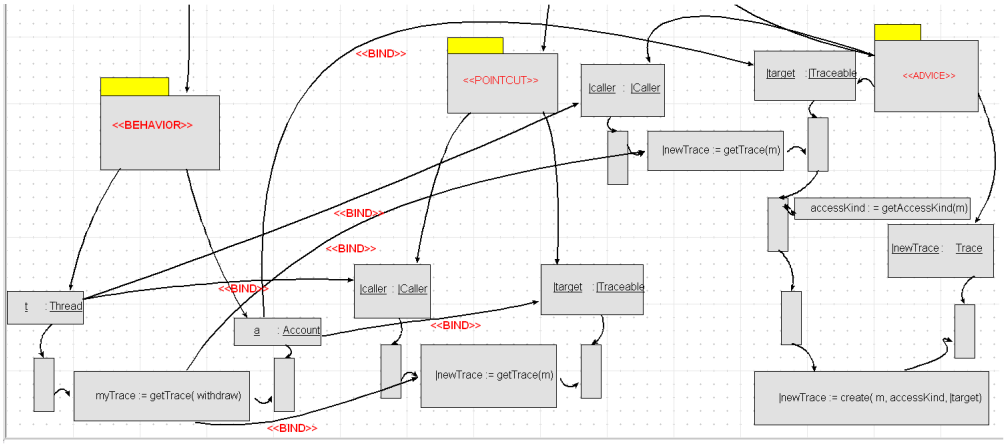

Fig. 14. Second part of the binding amongst the aspects modeled in AToM^3

3.2 Graph Rewriting

AToM^3[2] uses a LHS(Left Hand Side)/RHS(Right Hand Side) rule based graph grammar to perform the graph rewriting. The tool allows one to specify rules and assign them priorities. Each rule is composed of a LHS and a RHS, in the LHS one can specify the graph or sub-graph they wish the tool to detect and in the RHS you specify the result. In the RHS you can specify how the LHS gets rewritten or you can even completely replace the LHS with a completely different graph. Once the rules have been established you generate an executable of them using the tool and then run it on the model. The rules will run starting with the highest priority until no matching graph or sub-graph as specified on the LHS can be found. Once this occurs it moves onto the next priority rules and repeats until no more rules can be executed.

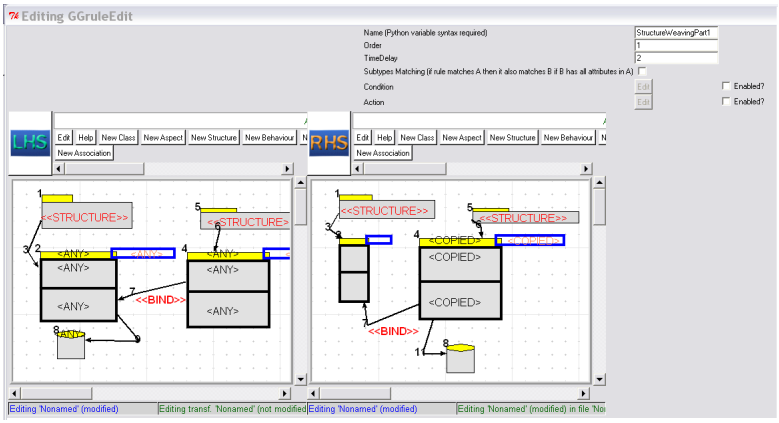

Fig. 15. First rule for weaving class diagrams in AToM^3

For this paper, to implement the weaving for the example in Fig.13 and Fig.14, we have specified 16 rules. We shall only present the important ones in detail. A lot of the rules have similar functionality in terms of the output they produce on the RHS but simply detect a variation of LHS graphs. Also, some of the rules were put there as a means of performing cleanup of the model but are not integral to the weaving itself. Fig.15 shows one of the rules for weaving class diagrams within aspect models. The goal of this rule is to find all associations from the bound template class and move these associations to the template class that initiated the binding. Similarly there is another rule(not shown here) that will take associations to the bound template class and move them to the template class that initiated the binding. Also, there is a set of two rules that will move all classes not associated with the bound template class to the structure of the aspect that initiated the binding. This is with accordance to how class diagrams get weaved, in that it gets merged within the aspect that initiated the

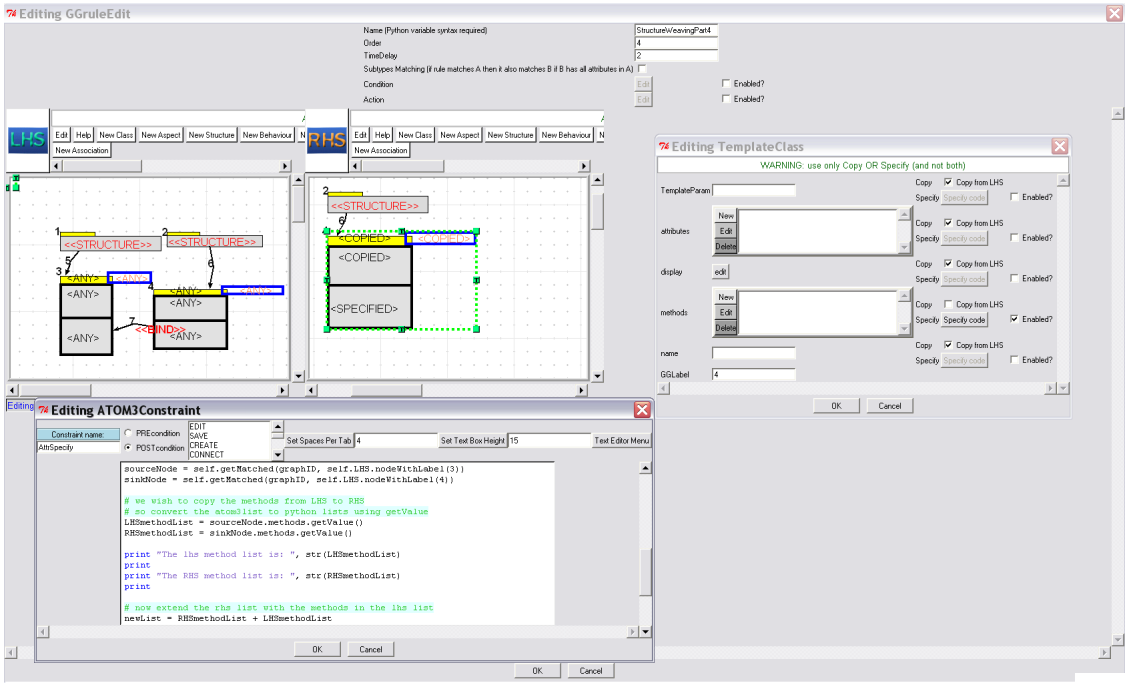

Fig. 16. Final rule for weaving class diagrams in AToM^3

binding. Fig.16 Shows the final rule for class diagram weaving between aspects and this is the rule where the methods of the bound class get appended to the methods of the template class that initiated the binding. This rule has the lowest priority of all rules related to class diagram weaving hence you will notice that in the RHS we get rid of the template class indicating that the merging has been finalized. Also this is a rule where python code needs to be specified in order to perform this appending. It is a rule composed of two parts, first graph rewriting is used to detect the sub graph pattern and then on the right hand side in the methods attributes of the template class as shown in Fig.16, code is specified as a constraint to copy over the methods. The next set of 5 rules are specific to class diagram weaving between the base aspect(our application) and the aspect Traceable after AccessClassified was weaved in. The rules are the same as those above except that the class that initiates the binding is not a template class but a regular class hence on the LHS it detects binding between a class and a template class.

Fig.17 shows a rule for sequence diagram weaving it will detect binding between two objects and perform a string replacement of the names of the bound object with the class and obj name of the binding object. Again to perform any kind of name change this is done via python code by defining a constraint as

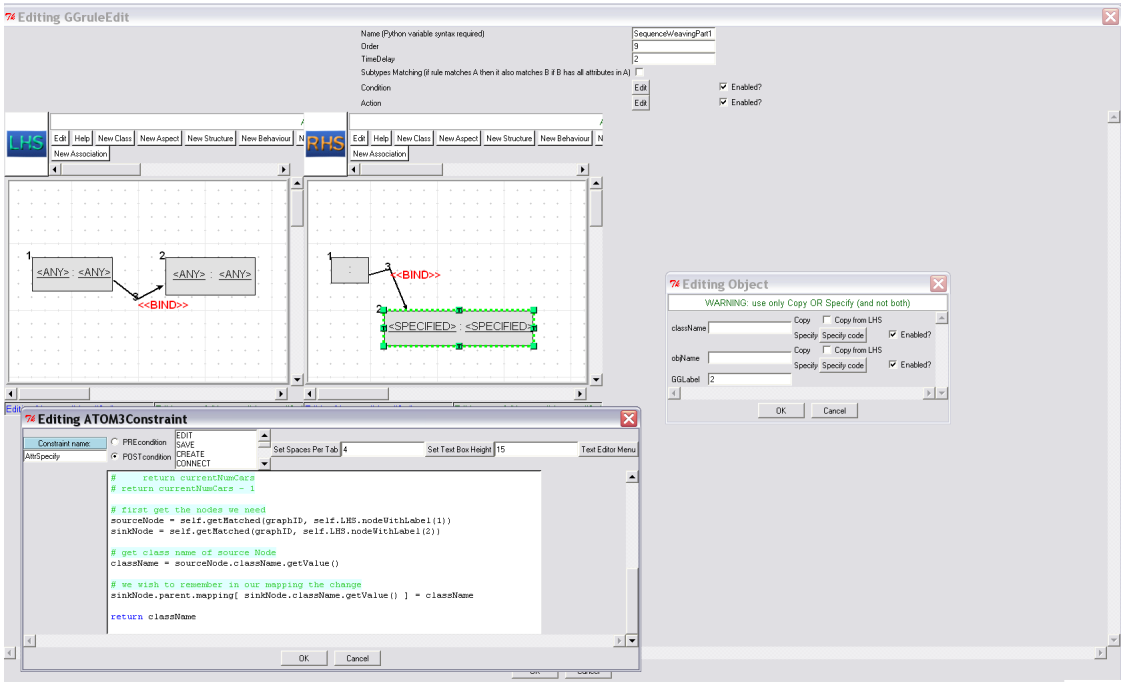

Fig. 17. First rule for weaving sequence diagrams in AToM^3

shown in the figure. Similarly Fig.18 shows a rule for sequence diagram weaving it will detect binding between two message objects and perform a string replacement of the message name of the bound object with the message name of the binding message object. Again to perform any kind of name change this is done via python code by defining a constraint as shown in the figure. Fig.19 shows the rule that will initiate replacing the behavior by the advice. It will use the binding associations defined to match at the LHS which advice to merge and where to merge and will begin connecting objects, formerly associated with the advice, to the base behavior. The second part of this rule as shown in Fig.20 will simply replace the binding object with the bound object and finally delete the binding object. With all these rules established we can execute these rules in AToM^3 and get a resulting final weaved model shown in Fig.21.

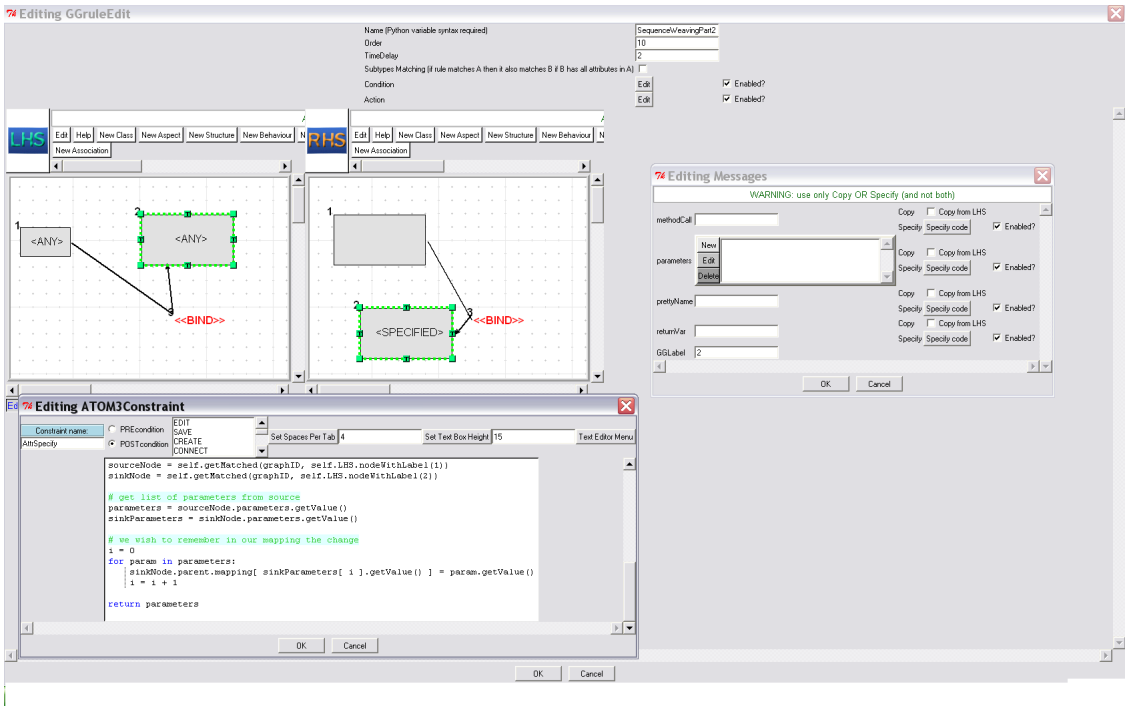

Fig. 18. Second rule for weaving sequence diagrams in AToM^3

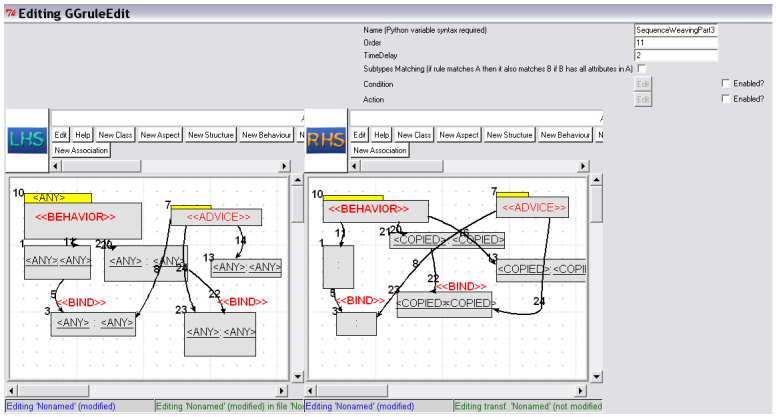

Fig. 19. Third rule for weaving sequence diagrams in AToM^3

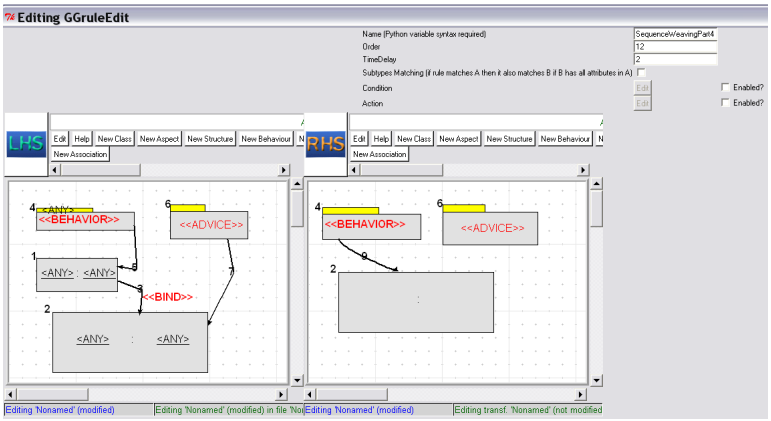

Fig. 20. Fourth rule for weaving sequence diagrams in AToM^3

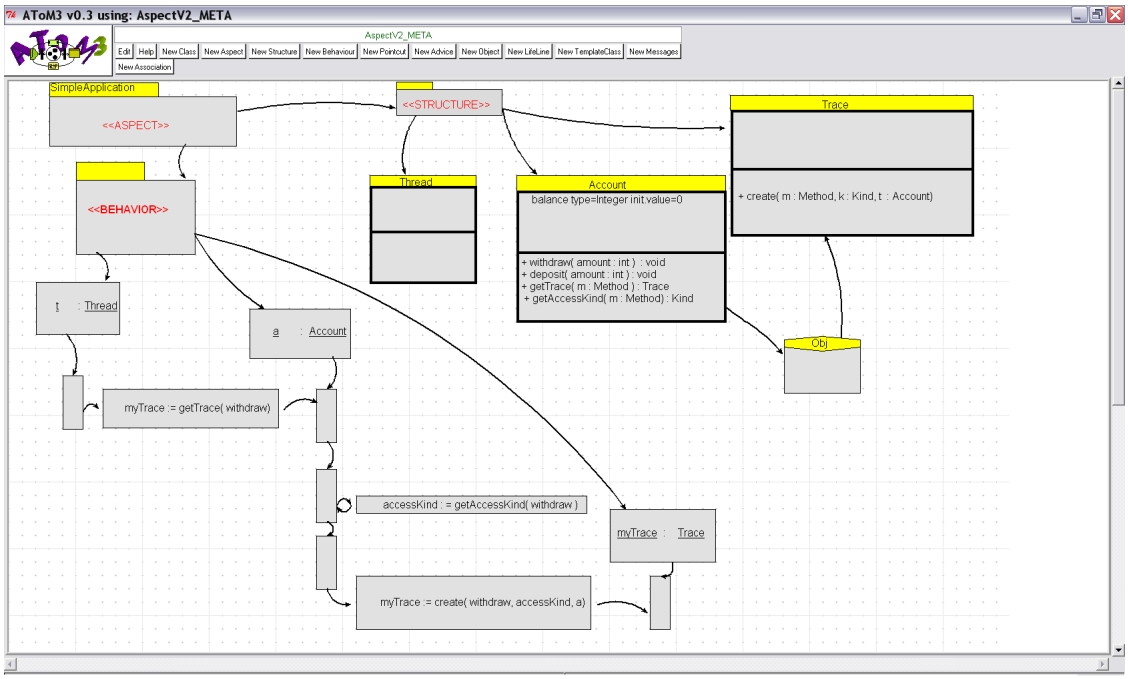

Fig. 21. Final Weaved Model after executing the transformation rules in AToM^3

4 Limitations

There are some important issues and limitations to the implementation presented in this paper that need to be discussed. The most crippling issue was the inability to group rules together and assign them to a region in the model. When you have an aspect dependency chain as seen in Fig.3 the weaving must be performed at the lowest level(if you imagine the dependency chain as a tree then we start at the leaves) and recursively move up weaving as we go along until we reach our base application and then perform the final weaving. In the context of our example the rules where set up so that the aspect AccessClassified gets weaved into Traceable first then the resulting intermediate aspect gets weaved into our base application. However, with graph rewriting and the way it is set up in AToM^3[2] the matching of the LHS in a rule can occur anywhere in the model and is not tied to a particular region. Hence, this would mean that we could start weaving at multiple levels since the execution of rules occurs randomly. Instead, the desired functionality is that the group of rules that perform the weaving must be applied to a specific region until they cannot be applied anymore. Next we define a new region to apply the rules and start over until we have our final weaved model. In order for this implementation to be tested on more complex examples this must be overcome.

The current meta-model shown in Fig.8 and Fig.9 is incomplete in that it was designed to the bare minimum just enough to be able to implement the example presented in this paper. This leaves much room for improvement, in particular, if one wishes that the class and sequence diagrams be compliant with UML 2.0. Furthermore, we noticed that some of the rules implemented required introducing python code in the constraints in order to properly change names of object, classes, messages and parameters. This could be heavily minimized if AToM^3[2] modeled everything. In the meta-model we could have modeled everything including parameters. This would have made binding more verbose but as a result it would have allowed us to perform weaving completely using graph rewriting which is the true desired goal. Finally given that this implementation was tailored to implement this simple example this means it is too specific and not generic enough to accomodate weaving for any aspect model. The ultimate goal is to define the rule set and meta-model in such a manner that any aspect model that can be represented using the techniques presented by Kienzle et al.[1] can be modeled in AToM^3.

5 Future Work

Some of the interesting and important future work to do is to first and foremost improve and enhance upon the meta-model. This can be done on many levels, one could expand on the class and sequence diagram representations and make them more expressive such as introducing alternative branches and loop constructs for sequence diagrams. Another possibility is to model parameters and attributes in classes as opposed to leaving them stored internally as attributes of the class model. Establishing constraints to perform checks such as

unique naming and so on is also important. On a lesser note we should eventually make drawing these models visually more appealing and user friendly. The most critical work to be done is to improve upon the weaving by introducing the notion of grouping grammar rules in order to be able to perform weaving recursively starting from the innermost aspect. Another interesting area to research is integrating OCL with this aspect modeling approach. Furthermore, from the perspective of the implementation in AToM^3[2], we should look at implementing conflict resolution aspects as presented in the paper[1]. Another interesting area of research would be to define some way of testing the weaving for potential errors. Finally, it would be nice to examine weaving of other behavioral modeling notations such as state diagrams.

6 Conclusion

In this paper, we have presented a brief and simplified overview on how Kienzle et al.[1] built an aspect-oriented modeling framework. We also touched upon how they combined feature modeling techniques with their AOM approach, as a way of managing variability in their framework. We discussed how the derivation of a transaction is performed by weaving the inter-dependent aspect models involved in the aspect dependency chain obtained from the product line. We have also shown how their weaver of reusable aspect models supports the modeling of structure (using UML class diagrams) and behavior (using UML sequence diagrams). Furthermore, we presented one implementation of a subset of this approach in a meta-modeling tool AToM^3[2] and applied it to a specific example. We examined how a meta-model can be created to allow one to model such aspects. Next, we examined how we can use graph rewriting to perform the weaving. Finally, we discussed some of the limitations to this implementation and the necessary future work that needs to be done.

7 Acknowledgments

I would like to thank Professor Jorg for all the help and insight he provided regarding the AOM technique he developed. I would also like to thank Professor Hans for all the time and effort spent helping with regards to the implementation in AToM^3[2]. Finally, I would also like to thank all the students of Comp 763 and in particular Reehan Sheikh and Raphael Mannadiar for the help they provided in using AToM^3.

References

1. Kienzle, J., Klein, J., Guel, N.: Modeling Aspect-Oriented Framework: Handling Aspect Reusability, Dependencies, Variabilities and Conflicts. Transactions on Aspect Oriented Software Development (TAOSD) (2008)
2. AToM^3: A Tool for Multi-formalism and Meta-Modeling. (`http://atom3.cs.mcgill.ca/`)