UML Model Refactoring

Viktor Stojkovski

University of Antwerpen, Faculty of Computer Science, Master Studies - Software Engineering, Antwerpen, Belgium

Abstract

Creating a complex UML statechart diagrams from a scratch can lead to some bad designs which afterwards in order to be changed and modified to a better functional and more compact model can be a painstaking job. The reason for that is the already complex statechart which is like a chain, each small change in the design can lead to numerous small changes in some other parts in order for the model to maintain its functionality. There is also another problem with the non-automated statechart transformations and that is, the designer cant be completely sure that after all those modifications the functionality of the statechart will remain unchanged. In order to prevent those problems from happening, in this project I will present and explain how I have implemented and used some already defined statechart transformations published in the scientific paper with a title Refactoring UML Models Sunye et al. (2001), with the help of the software AToM3. AToM3 is a tool mainly used for meta-modeling and model transformations, which helped me a lot to implement and test the rules used for refactoring the statecharts. The rules for refactoring are made from preconditions and actions, which guaranty that the functionality of the statechart model will be unchanged after its transformation. In the end after using the known transformations on some test models, the final product is good structured and designed UML statechart Selic (2009) model which has preserved his behavior.

Email address: Viktor.Stojkovski@student.ua.ac.be (Viktor Stojkovski)

1. Introduction

The main task for this project is using the already defined refactoring rules in the paper Refactoring UML Models by Authors and the constraints for pre and post conditions defined with the OCL (Object Constraint Language) to create rules in AToM3 `http://atom3.cs.mcgill.ca/` using the Graph Grammar. For the purpose of fulfilling the defined pre and post conditions for every rule, I have used conditions and actions on almost every created rule. The defined rules work in a way such that, they are searching for patterns in the created models (in this case UML statecharts) and after recognizing a pattern they try if the conditions are true and if its so they execute the defined actions for that particular rule. The problems with the defined refactoring rules are that in the paper they are only defined theoretically, but not in practice. The conditions written with the OCL language OMG (2009) are using a statechart meta-model which was not provided with the paper, but also by studying the OCL constraints one can realize that the meta-model used for those rules is different than the meta-model used in AToM3. The challenge is to try to transform and implement the refactoring rules and also try to reproduce the results from the experiments that are only theoretically approved. In order to achieve that aside from defining the main rules for refactoring there must be created a few smaller rules which will follow the other in order to refine the statechart mode in my case, deleting transitions and contains Hyperedges. AToM3 has provided me with an already done meta-model for statecharts called DCharts which are explained in the paper DCharts, A Formalism For Modeling And Simulation Based Design of Reactive Software Systems Feng (2004) and the same MM (Meta-Model) I am using as an formalism, because of the need to change some attributes of the DCharts model.

The next section, Section 2 is providing the reader with some useful information about the related work and in this case that is the paper Refactoring UML Models Sunye et al. (2001). Section 2 will provide a summarization of that paper and also will include some comments about parts of the text and brief explanation of some of the created rules for refactoring as well as my conclusion on the article, which has helped me and provided very useful information for starting my project and during the process of creating the GraphGrammar (GG) `http://atom3.cs.mcgill.ca/people/jlara/AToM3_Programming/GraphGrammars.dtml`.

Section 3 will contain an explanation of the working process, how I have implemented the rules, problems during the implementation and some idea about extending the work, because what has been done in this project is just a starting point and a guideline for allowing a further extension and upgrading the already defined and implemented rules. Statecharts are a large field for exploration which opens many possibilities and occasion for continuing the work in this field. The Section 3 will be the main part of this dissertation and will include a number of smaller sections explaining the parts that were mentioned in this paragraph.

In Section 4 there will be presented some of the results obtained by using the GG (GraphGrammar) rules on some simple statechart models and a few more complex models. In this sections the reader will be presented with the final results of the project and a conclusion on whether it has been a successful experiment and what are the advantages by using this, for now rather simple refactorings.

Section 5 will be reserved for my conclusions on this project, what are the advantages/disadvantages of using the newmade refactorings and also what has been my personal gain from working on this kind of project related to the field of Model Driven Engineering.

2. Summarization of the article Refactoring UML Models Sunye et al. (2001)

This paper is oriented on a refactoring UML models, in particular the examples used are for UML Class and Statechart Diagrams. Because of the necessity of software evolution, the software designers are in constant need of already predefined tools for transformations. Those transformations will ensure the software engineer that there will be no changes in the behavior of the software, but only the structure of the software will be changed in order of better functionality and continuous improvement. The transformations can also be called refactorings, which by definition should be behavior preserving transformations of an application.

The UML is well suited for testing these kind of transformations because

of the defined meta-model, which can be used to control the modifications, but in the same time keeping the functionality of the application unchanged. In order to keep the behavior of the software the same, the usage of OCL (Object Constraint Language) constraints at the meta-model level is required. The authors of this paper are making an assumption that if we control the impact of the modifications, it should be possible a small set of transformations to be constructed and from that set, design patterns can be made which can be applied on some UML models for their refactoring. In my project those design patterns are constructed as Graph Grammar rules, which have two sides LHS (Left Hand Side) and RHS (Right Hand Side). On the LHS I present the pattern for which the grammar should search and on the RHS are defined the refactorings which should be applied on the graphical appearance and on some of the attributes of the used entities. These rules and their actions will be more explained in Sections 3 and 4.

As a first transformation example is used a small Class Diagram refactoring example from which are made a few conclusions. Using the Composite design pattern they are trying to restructure the class diagram without changes in his behavior. Applying that pattern has made some changes in the class diagram, which are then justified one by one. For example, a generalization has been made between two classes which does not change the behavior of the model because the generalized class does not have superclass and the general class is empty (without attributes). For refactoring class diagrams there are 5 basic operations used: addition, removal, move, generalization and specialization of the modeling elements. All of these refactorings are described in the few following chapters, but I will not explain them in details because the Class Diagram refactorings are not included in the working field of my project and they are not very important for the matters of UML Statechart refactorings.

More complex and also more elaborated is the second part of the article, where is presented an example which is showing some refactorings used over the statechart diagram model. Working with statecharts is more difficult because they dont model only the structure of the system like the Class Diagrams, but also they model its behavior. In the paper as an example a model of a phone state diagram is used and some of the refactorings are tested on this model. In Section 4 can be seen the results of refactoring the phone model using the implemented transformation rules in AToM3 GG. Over this particular model 4 transformations are executed and are justified in order to be shown that after the refactorings the functionality of the di-

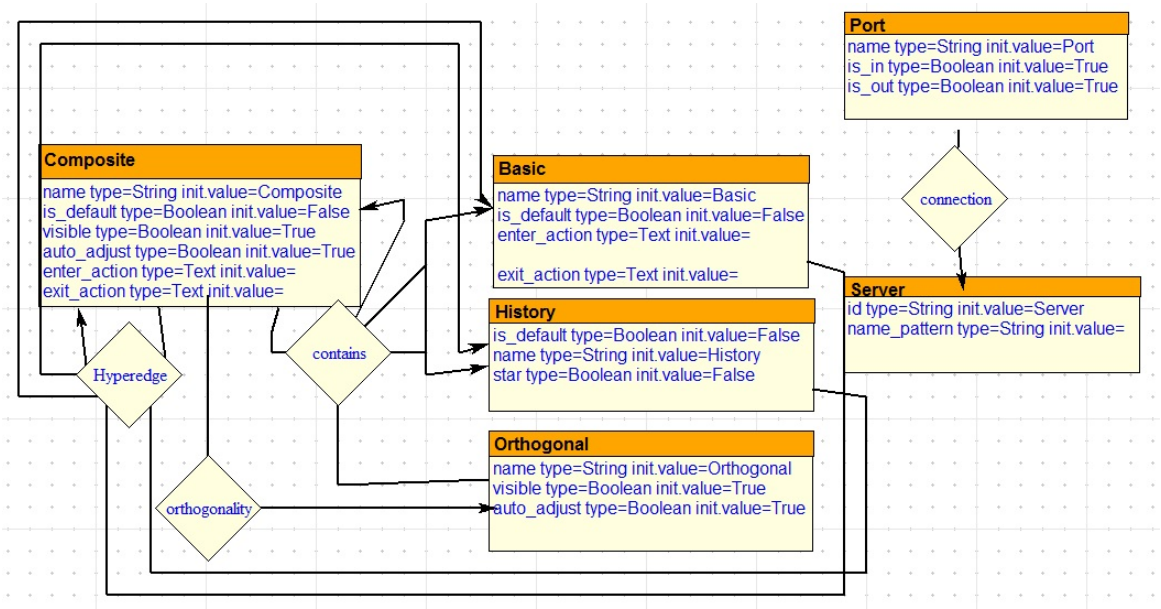
agram is the same. First transformation that is used is **Group States** and it creates one Composite State which includes all of the states in the statechart. Using this refactoring rule doesn't harm the functionality, but only changes the hierarchy in the model. The second transformation is called **Move State Out of Composite** and what it does is moving the state Idle and the initial state out of the composite state Active. That is legal because the newly created composite state named Active doesnt have entry or exit actions and because of that the order of execution of the existing actions is unchanged. For making refactoring rules for the statecharts work correct, constraints must be used and those comnstraints must be satisfied before and after each transformation for the sake of behavior preservation. For creating the constraints the language OCL is used at the metamodel level of statecharts. Most complex parts for refactoring are the states and the composites which have attached some of the actions as do, entry and exit. Each of these actions is executed in particular time, as an example the action do is executed when its state is active. The description of refactorings for the statecharts is divided in two parts: refactorings concerning the state and refactorings over the composite state. The first two transformations are called **Fold Incoming/Outgoing Actions** and their job is to replace a multiple set of actions attached to transitions which are incoming/outgoing for a particular state by an exit/entry action attached to the state. For every transformation there are some restrictions that need to be considered and for this particular transition the action must be equivalent in order to be folded and the transition to which they are attached must not cross a boundary of a composite, also the state mustnt own any entry or exit actions. Concerning the transformations of a state there are a few more of them explained such as **Unfold Entry/Exit Action** (symmetrical transformations as the previous ones) and Group State (groups one or many states in one composite state), all of them are briefly explained and there is a particular OCL constraint attached to them. The second part of the statechart refactoring rules is describing the transformations which can be done over the composite state. One such is the **Fold Outgoing Transitions** transformation, it replaces the set of transitions leaving the components (states) of one composite state with a single transition. There are few restrictions before this refactoring can happen and those are: all the transitions must lead to the same state and the folded transition will lead to that state and all the actions attached to the transitions, if there are any, must be equivalent. There are a few more complex transformations such as Move State into/out of Composite, which

have much more complex constraints than the first one and this particular refactoring rule will be explained in Sections 3 and 4.

In the conclusion there are mentioned a few experiences of refactoring activity diagrams and some other difficulties including the problem with the not very precisely defined abstract syntax of OCL and because of that they didnt managed to test their constraints or improve the definition of some of the refactorings. At the end of this paper, the authors are mentioning an extensive use of the action semantics in the future in order to make models more precise and more secure refactoring although the action semantics for the UML language are only standardized in a natural language (English) as it is stated in the paper Formal Action Semantics for a UML Action Language Yang et al. (2008).

3. Design of the solution and the work process

Before I start to explain all the rules separately i would like to present the design of the meta-model of the Dcharts which are considered as a substitute for the statecharts. On the picture below it can be seen the model which i'm using as a formalism in order to create new DChart models, but also it is possible to make some changes in the formalism of the DCharts. As an example a have added a boolean attribute called toDelete to the Hyperedge which is the transition, and that attribute marks the transition whether it needs to be deleted or not. This subrule will be explained in the folowing chapters. In order to create the meta-model for the DCharts there must be also a meta-model for the DChart formalism. The DChart model is created in a ER Diagram (Entity-Relationship), which consists of 3 types of building blocks: entities (square blocks), relationships which are connecting the entities and attributes which can be assiciated with the entities and the relationships.



3.1. State Reafactoring Rules

3.1.1. Fold Incoming Actions

This refactoring is quite simple, what it does is, it checks if all the incoming transitions to one state have an actions and if those actions are equal

one of the conditions for executing the refactoring is checked. Two actions are equivalent if their operations have the same effect or if they send the same signal. Another constraint for this rule is that the state for which the transitions are incoming doesnt have an entry action, because if all of the preconditions are satisfied two thing are going to happen. First all the actions of the transitions will be deleted and second the equivalence of those actions will be added to the field entry action of the state. In addition i have also added a new preconditions which says that the refactoring should be executed only if the number of incoming transitions is larger than one. If this constraint was not added the actions after the refactoring of the rules about the states will start to travel through the diagram which is not good and will completely mess up the statechart desigh. Under this paragraph the code of the precondition is presented. I have created a small number of functions which are often used as a part of the preconditions and action. Such are as in this function **compareTransAct** which gets two arguments, a state and a string which says which transitions should be taken in consideration, incoming or outgoing.

```
# condition for the refactoring fold incoming actions
def foldIncomingActionsCond(self, graphID, stateLabel):
    state = self.getMatched(graphID,
        self.LHS.nodeWithLabel(stateLabel))
    enterAction = state.enter_action.toString().strip()
    # compare if all the input transitions actions are
        the same
    allInTransActSame = compareTransAct(state, "in")
    nrInTrans = countTrans(state, "in")
    empTrans = emptyTransition(state, "in")
    # are the same, the number of transitions is
        larger than 1 and the actions are not
    # empty fields
    if not enterAction and allInTransActSame and
        nrInTrans > 1 and not empTrans:
        return 1
    else:
        return 0
```

In this paragraph will be presented and explained the action code for the fold incoming actions rule. First the value of the action from some of the incoming transitions is taken and the state attribute enter_action is initialized with that value. After doing that what is left is to pass to all of the incoming transitions from another states and delete their actions using the function deleteActFromTrans which as an value gets an array and deletes all the actions from the transitions.

```
def foldIncomingActions(self, graphID, stateLabel):
    state = self.getMatched(graphID,
       self.LHS.nodeWithLabel(stateLabel))
    action = getActFromTrans(state, "in")
    transitions = []
    state.enter_action.setValue(action)

    for trans in state.in_connections_:
        if isinstance(trans, Hyperedge):
            for checkState in trans.in_connections_:
                if isinstance(checkState, Basic):
                    transitions.append(trans)
    deleteActFromTrans(transitions)
```

3.1.2. Fold Outgoing Actions

This refactoring is very similar to the previous one and because of that it will be only explained with text, without supporting code. As the previous rule this transformation is not very complex. Its precondition is, the outgoing transitions from the state on which the refactoring is to have actions and all of them need to be the same. Another condition is that the state should not have an exit action and as last, the number of transitions outgoing from that state should be larger than 2. The precondition of this rule is applied to every state in the statechart, but the rule will be executed only on those states which will fulfill the requirements. The actions taken by this rule are very similar to the previous one. It assigns the value of the action of the transitions to the exit action field of the state and it deletes all the values of the action attributes of the outgoing states.

3.1.3. Unfold Entry Action

This and the next refactoring rule are completely opposite from the previous two rules. Unfold Entry Action is symmetrical rule to the Fold Incoming Actions and what it does as a precondition is, checks if the state has a value assigned in the field entry action and it copies to the action attribute of the incoming transitions of that state. In order for that to happen the incoming transitions mustn't have an actions and also their number has to be larger than 2. Below this paragraph the action code of this rule can be seen. With the predefined AToM3 functions in_transitions_ and out_transitions_ we move through the diagram and check if some of the conditions are fulfilled. Also with the function setValue() the value of some attribute can be set by putting the contents inside the brackets, as it can be seen below that I'm setting the value of the attribute enter_action of the state to an empty string.

```
def unfoldEntryAction(self, graphID, stateLabel):
    state = self.getMatched(graphID,
       self.LHS.nodeWithLabel(stateLabel))
    enterAction = state.enter_action.toString().strip()
    transitions = []

    for trans in state.in_connections_:
        if isinstance(trans, Hyperedge):
            for checkState in trans.in_connections_:
                if isinstance(checkState, Basic):
                    transitions.append(trans)
    addActToTrans(transitions, enterAction)
    state.enter_action.setValue("")
```

3.1.4. Unfold Exit Action

This rule is working in a similar way as the Unfold Entry Action, but instead of working with incoming transitions and entry action, it checks if the state has a set value of the attribute exit_action and if all the outgoing transitions dont have any actions. Also the number of outgoing transitions should be larger than 2 so the actions can be performed. The refactoring consists of deleting the value of the attribute exit_action of the state and on every outgoing transition action field assigning the deleted action from the state.

3.1.5. Group States

This is the first more complex refactoring in this paper and because of that the condition and the action code will be explained. Below, the code for the precondition is shown and what it does can be also concluded from the comments in the code. In short, with the command state.roodNode.listNodes['NodeName'] all the nodes in the AToM3 model with the specfied name NodeName can be taken and putted in a list. If the number of the composites in the model is larger than 1 and if there are states on the root level of the model all of them will be contained in the new created composite state.

```
def groupStatesCond(self, graphID, stateLabel):
    state = self.getMatched(graphID,
       self.LHS.nodeWithLabel(stateLabel))
    # get all the states from the model
    states = state.rootNode.listNodes['Basic']
    # get all the composite states from the model
    comps = state.rootNode.listNodes['Composite']
    # if there are no composite states in the model
       check if there are any states
    if len(comps) is 0:
        if len(states) > 0:
            return 1
    # if there are any composite states in the model
       check if there are any states
    # out of the composites and if it's so return
       positive answer
    elif len(comps) > 0:
        for st in states:
            if not getCompositeFromState(st):
                return 1
        return 0
```

After fulfilling the requirements there are a few actions that need to be done. As it can be seen from the action code below with the function createNewComposite() a new composite state is created. Now the composite needs to be connected to all the states and composites in the top of the model hierarchy and that is done with the helping function makeConnection(), which takes two nodes and connects them. With this rule I have explained all the

implemented rules that are concered with the states in the statechart model, next to be explained will be the refactorings for the composite.

```
def groupStates(self, graphID, stateLabel):
    state = self.getMatched(graphID,
       self.LHS.nodeWithLabel(stateLabel))
    states = state.rootNode.listNodes['Basic']
    comps = state.rootNode.listNodes['Composite']
    # create a new composite state
    newComp = createNewComposite(state.parent, 100,
       100, 1)
    # for all the states that fulfill the conditions
       connect them with the new composite
    for st in states:
        if not getCompositeFromState(st):
            makeConnection(newComp, st)
    # for all the composites that fulfill the
       conditions connect them with the new composite
    for comp in comps:
        if not getCompositeFromState(comp):
            # don't connect the new composite with
               itself (atom3 was doing that)
            if comp == newComp:
                continue
            makeConnection(newComp, comp)
```

3.2. Composite Refactoring Rules

3.2.1. Fold Outgoing Transitions

This refactoring is more complex than the other rules explained so far. Below is the code for checking the preconditions before executing the actions and from the code comments it can be clearly understood what it does. For the sake of clarifying the code actions it will be briefly explained. What this transformation does is, it's checking if all the states in one composition have mutual outgoing state, it deletes the transitions and creates new Hyperedge from the composite to the destination state. Here are used a few helping functions such as getAllStatesFromComp(composite) - it gets all the states inside the given composite, checkDestState(statesList, destState) - it

checks if the given destState is a destination state for all the states in the list statesList, incomingTransConnection(statesList, destState) - it returns a list of transitions between the states in the list and going to the destination state and equalTransAct(transitions) - it recieves as an input a list of transitions and returns true if their actions are equal.

```
def foldOutgoingTransitionsCond(self, graphID,
    stateLabel, destStateLabel):
     state = self.getMatched(graphID,
         self.LHS.nodeWithLabel(stateLabel))
     destState = self.getMatched(graphID,
         self.LHS.nodeWithLabel(destStateLabel))
     comp = getCompositeFromState(state)
     compDestState = getCompositeFromState(destState)
     # checks if the state is in a composite and also
         if the destination state composite, if any,
     # is different from the source composite from
         where the folding will be created
     if comp and comp <> compDestState:
          compStates = getAllStatesFromComp(comp)
          # checks if all the composite states are
              connected with the destination state
          if checkDestState(compStates, destState):
               # get all the transitions between the
                   composite states and the destination
                   state
               transitions =
                   incomingTransConnection(compStates,
                   destState)
               # check if all the actions of the
                   transitions are equal
               if equalTransAct(transitions):
                    return 1
     return 0
```

After the checking of the conditions there are a few actions that need to be completed. First the newly created attribute in the Hyperedge called toDelete is set to true because the transitions from the composite states

and the destination state need to be deleted. The delete process will be done by another rule called deleteTransitions which will be explained in the next sections where the results of using the rules will be presented. After the deleting of the transitions a new connection should be made between the composite and the destination state and that is done with the function makeConnection().

```
def foldOutgoingTransitions(self, graphID, stateLabel,
    destStateLabel):
     state = self.getMatched(graphID,
         self.LHS.nodeWithLabel(stateLabel))
     destState = self.getMatched(graphID,
         self.LHS.nodeWithLabel(destStateLabel))
     comp = getCompositeFromState(state)
     compStates = getAllStatesFromComp(comp)
     # set the toDelete attribute of the transitions to
         True
     setToDeleteAtt(compStates, destState)
     # create a new Hyperedge between the composite and
         the destination state
     makeConnection(comp, destState)
```

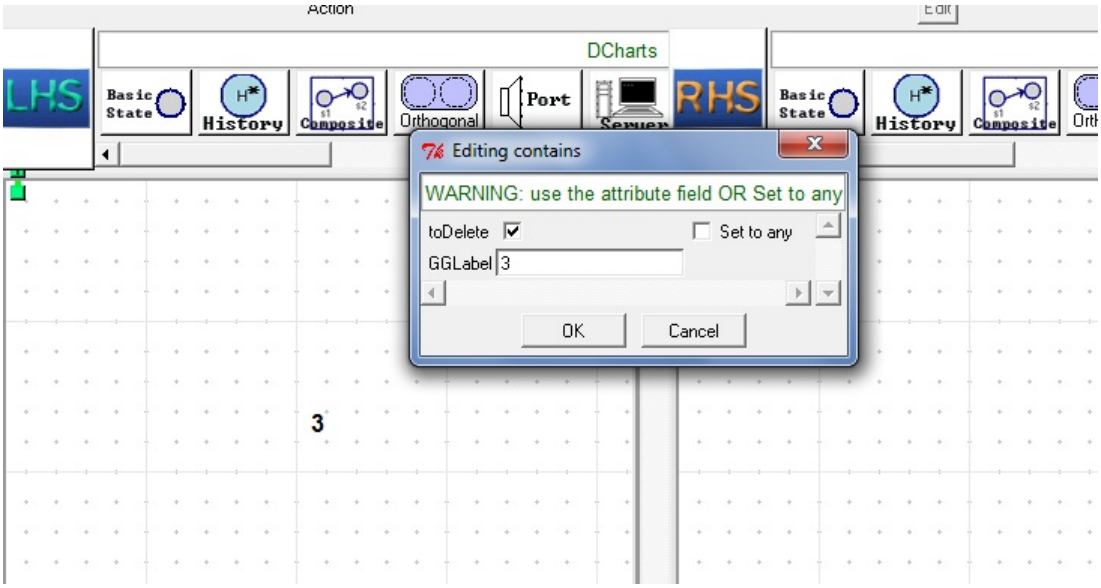
3.2.2. Move State out of Composite

This is the most complex refactoring rule implemented in this project and what it does it checks if some of the states can be moved out of the composite state and it takes the necessary actions for that transformation. The precodition needs to check a few more complex contraints. The first is if the state that needs to be moved out of the composite has an incoming/outgoing inner transitions (transitions that are not crossing the boundary of the composite) all the actions from those transitions need to correspond with the exit/entry actions of the composite state. If the composite has an exit/entry actions and the state is connected with transitions with outer states (states outside the composite), after getting the state out all the exit/entry actions of the composite should be added to the incoming/outgoing transitions of the state correspondingly. As last if the state is a default state for that particular composite all of the incoming transitions for the composite should be transfered to the moved state. The code for the preconditions and actions of

this refactoring is rather large, but after reading the previous explanations and the comments in the code the user should be able to easy figure out how the program functions.

3.2.3. Delete Contains

This rule is an example of a side rule for helping the other main rules to finish their tasks. The main task of this rule is to check if in the model there are any Contains transitions with the boolean variable toDelete set to True as it can be seen on the LHS (Left Hand Side), and it deletes them by leaving the RHS of the rule empty. The actual transition can't be seen on the LHS because of its properties to be not visual, but we can see its label which is the number 3.



3.3. Challenges and Room for Improvements

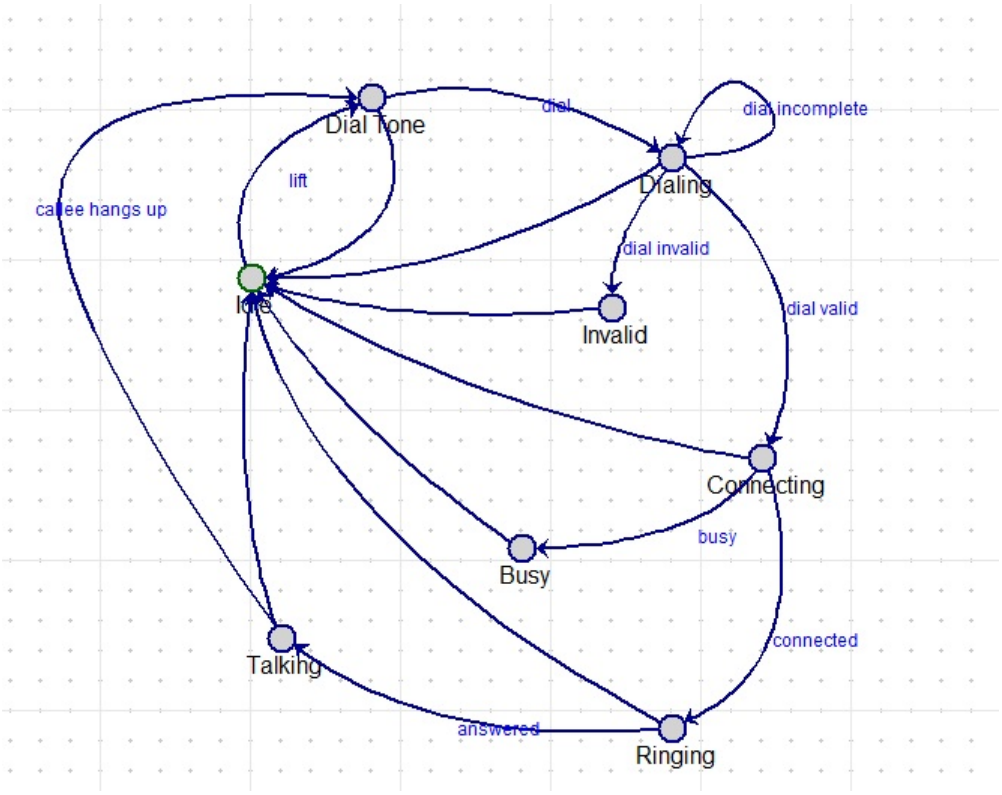
The biggest challenge was to implement the complex rules which have a need of checking all the possibilities for their execution. It was a common error during the testing that some of the rules which condition did returned false instead of true. There is also a room for a big improvement in the some of the preconditions of the refactorings. In the statecharts there are many possibilities and that is why the statechart can be in a lot of different states in different time. It is very important to cover all of the possibilities and

combinations that one state can be in, so that the precondition constraints will check only important conditions and will decide what should happen according to the given situation. The actions are pretty precise in their functions and explanation in the article by Sunye et al. (2001) and when executed they are refactoring only the things that theoretically we have established are needed to be refactored .Finding the errors in the code is also a challenging task because in one precondition there can be a lot of nested functions and the AToM3 doesn't point to the exact place where the error has happened, but that is solved using a number of prints positioned on a strategic points in the code. In the next section at the end it will be explained the new action a have added to the refactoring Move State out of Composite. That is when the state that is going to be moved out of the composite is a default state for that composite and if it has only one outgoing transition to some of the states in the composite. The rule will remove that transition and replace it with a transition from the state to the composite and the state to which the transition was going it will become a default state for that composition.

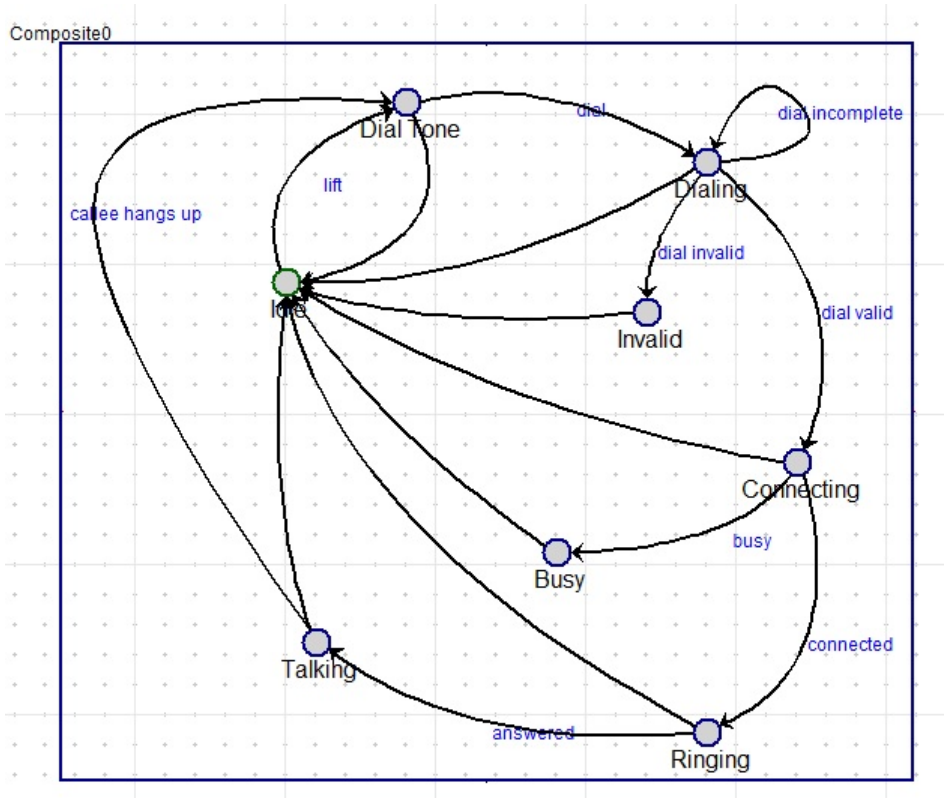
4. Using the created refactoring rules over statechart models

Within the AToM3 user area there are provided models for every implemented rule and they are named after the rule they are dedicated to. Those are small and simple models on which the refactoring rules are working properly. The most challenging experiment is done over the model presented in the article Refactoring UML Models Sunye et al. (2001) and that is the statechart model of the phone. After trying the theoretically correct transformations presented in the paper on the phone model, I have gotten the same results as expected. In the next few paragraphs the whole process will be explained and shown on pictures.

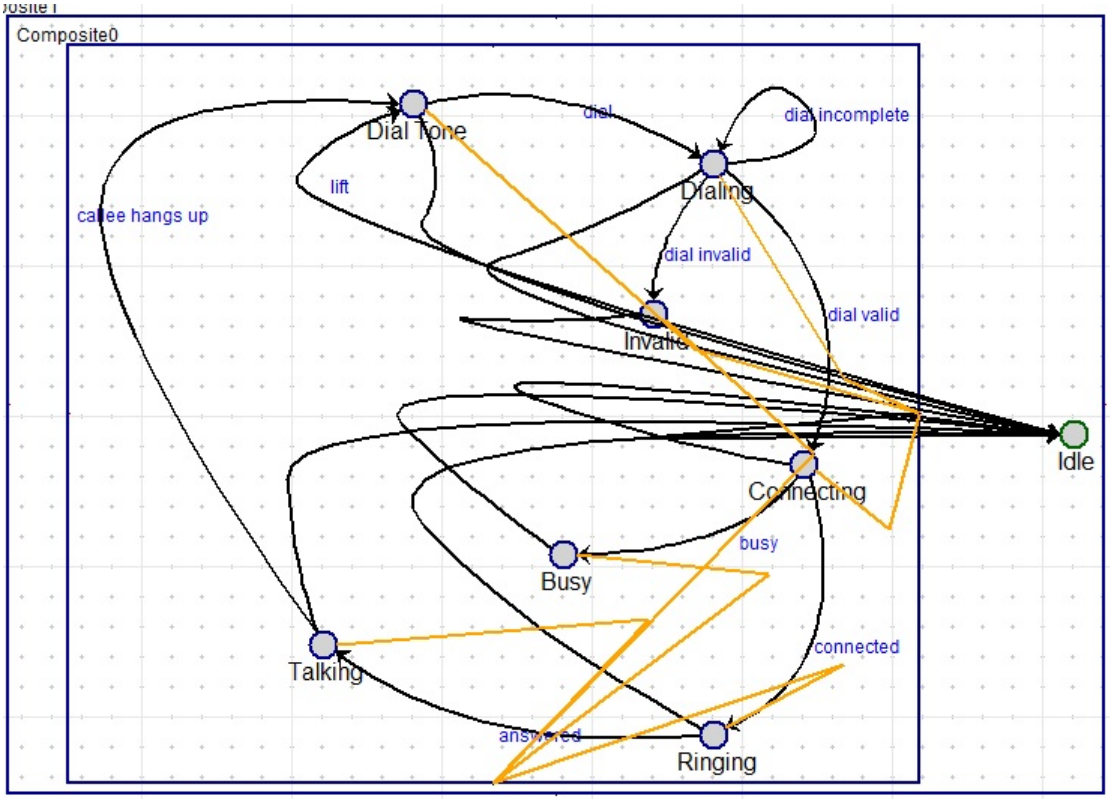
On the picture below the starting model of the phone diagram is presented. As it can be seen we can conclude that it is not the best modeled phone diagram. From every state there is a transition to the state Idle and that can be refactored so it will be more elegant and well structured without changing the model behavior.



The first rule with the biggest priority in the grammar is the rule group states which will group all the states into one composite and the result of that can be seen on the picture below. After grouping all states there will be good conditions for the next rules to be executed in order to get to the final result.

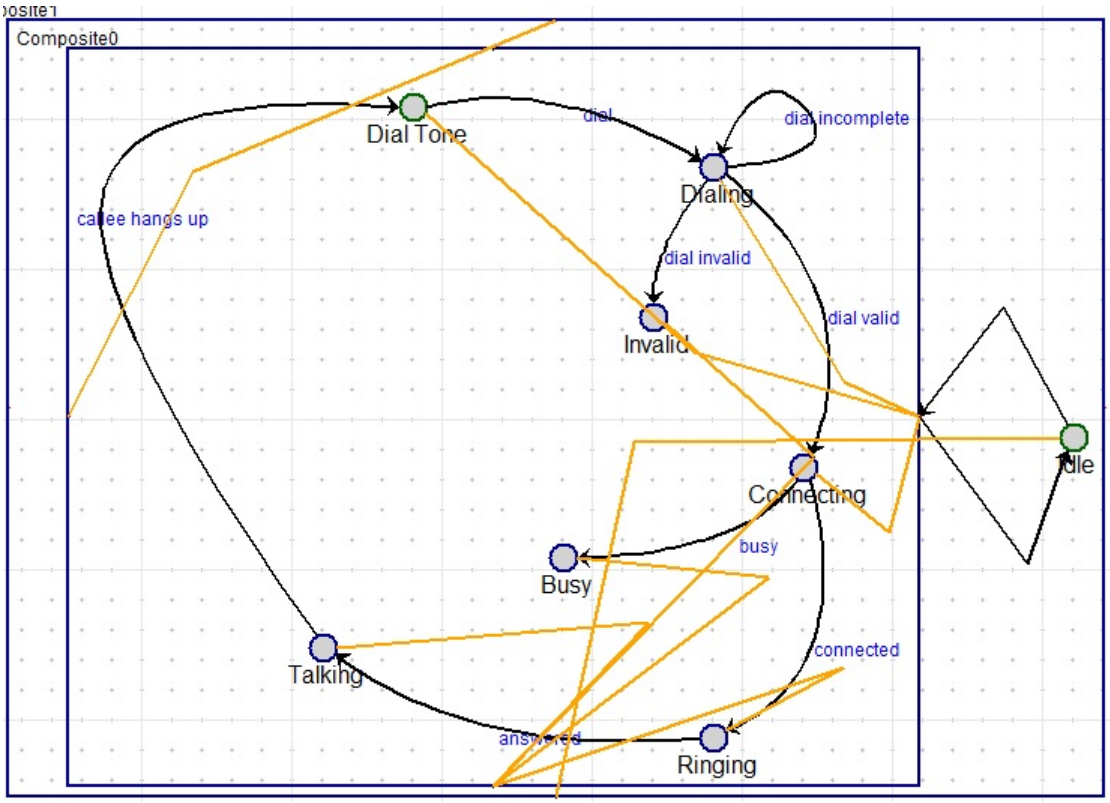


Next rule to be executed is the Move State Out of Composite refactoring rule. It should find the state Idle as a state that fulfills all the predispositions to be moved outside of the composite state. The effect of the refactoring can be seen on the following picture. Although the picture is not very clear it can be seen that the state Idle is taken outside from the Composite0, but all the transitions are making the model very messy and hard to understand. The next refactoring will take care of the bulk of transitions, but also it can be seen from the picture below that a new Composite1 is created and the Composite0 and the state Idle are inside it. That happened because the rule group states has the biggest priority and whenever it sees more than one state or composite on the highest hierarchy of the model it will group them in one composite.



In order to finish the experiment successfully there is only one refactoring to be done. That is the fold outgoing transitions rule which should fold all those incoming transitions to the state Idle into one transitions going from the Composite0 to Idle.

On the last picture below as it can be seen we can conclude that the experiment has been succsessful. All the transitions have been replaced with one transition, but there is also one property which has been explained in the section room for improvements and the result of that is, the state Idle is not connected to directly to the state Dial Tone. Instead the Idle is connected to the Composite0 and the state Dial Tone has been made a deafult transition inside the Composite0. With this example we can conclude that the refactorings were succsessful and that they give the expected results.


Composite0
Dial Tone
dial
dial incomplete
Dialing
callee hangs up
dial invalid
dial valid
Invalid
Idle
Connecting
busy
Busy
connected
Talking
answered
Ringing

5. Conclusion

It can be concluded after reading this project paper and after doing the implementation of the refactoring rules that the rules which were only theoretically proven that are working, now they can be proven also with a practical work. There are possibilities of correction and upgrading of the already implemented refactorings, but as a starting point in the field of practical refactoring of statechart UML diagrams, this project has achieved its goal. Also i can conclude that the method of Meta-Modeling as part of the Model Driven Engineering, has proven itself as a very powerful methodology for creating a hierarchy of meta-models and model and manage their behavior by using constraints, rules and predefined formalisms. AToM3 is also very helpful and powerful tool for better understanding and managing the meta-modeling levels and creating useful rules and conditions, without which the goal of this project won't be possible to be achieved. The future work on these UML Models Reafctoring rules can be in the direction of making them more sophisticated and useful also comprehending work on the complexity and throughness especially on the preconditions which are very important because they choose on which object in the diagram the rules will be executed.

References

Feng, H., 2004. Dcharts, a formalism for modeling and simulation based design of reactive software systems. A Masters Thesis Submitted in Partial Fulfillment of Requirements.

OMG, 2009. OMG Object Constraint Language (OCL). OMG.

Selic, B., 2009. Unified Modeling Language Specification (version 2.1). OMG.

Sunye, G., Pollet, D., Traon, Y. L., Jezequel, J.-M., 2001. Refactoring UML Models. Springer.

Yang, M., Michaelson, G. J., Pooley, R. J., 2008. Formal action semantics for a uml action language. Journal of Universal Computer Science.