Transforming Software Requirements by Meta-modelling and Graph Transformation

Ximeng Sun and Hans Vangheluwe

School of Computer Science, McGill University
Montreal, Quebec, Canada
{xsun16, hv} @cs.mcgill.ca

Abstract. This article describes transformations of software requirements. We start from requirements (use cases) in textual form. These are subsequently encoded precisely in UML 2.0 Sequence Diagrams (SD) by a modelling expert. The SD model is automatically transformed back into textual form to allow the requirements producer to check correct interpretation. Once the latter is satisfied, the SD model is transformed into a Statechart (SC) which satisfies the constraints specified by the SD model. This SC model can then be used to analyze properties of the requirements. This may involve adding information such as transition probabilities for dependability analysis, or even sufficient refinenment to allow for code synthesis. We use AToM^3 to meta-model both SD and SC formalisms. From this, visual modelling environments are synthesized. In addition, the transformations are modelled graph grammars. We demonstrate our complete approach in an example.

1 Introduction

We proposed a model-driven approach to transform software requirements. The model-driven approach starts with modelling requirements of a system in Sequence Diagrams, and the subsequent automatic transformation back into textual requirements or into Statecharts.

A visual modelling environment was built in AToM^3 using *Meta-modelling* and *Model Transformation* to support the model-driven approach. It supports modelling in Sequence Diagrams, automatic transformation to Statecharts, and automatic generation of requirements text from Sequence Diagrams.

Section 1 describes a meta-model for the Sequence Diagrams formalism. Using our visual meta-modelling environment AToM^3, both abstract syntax and concrete visual syntax are modelled. From this, a visual modelling environment for Sequence Diagrams is synthesized. A similar approach is followed for Statecharts. The meta-model is not given here. Section 2 describes model transformation mapping Sequence Diagrams models onto corresponding Statecharts. The transformation is modelled in AToM^3 in the form of Graph Grammars. A small example demonstrates the different phases of the transformation. Section 3 demonstrates how Sequence Diagrams can also be transformed into a textual requirements document for inspection by a stakeholder. Section 4 gives related work and a conclusion.

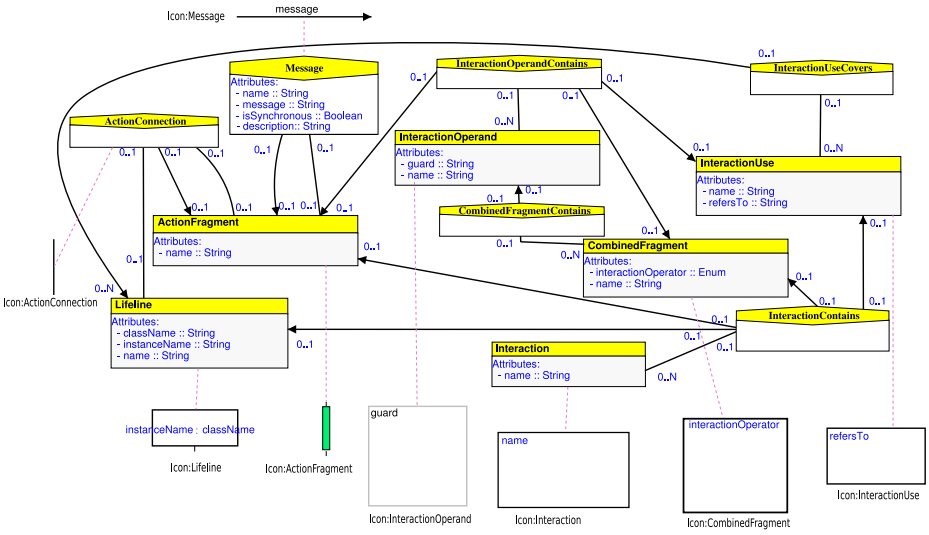

Fig. 1. Sequence Diagrams meta-model in the Class Diagrams formalism

2 Meta-modelling Sequence Diagrams

2.1 Sequence Diagrams Meta-Model

We start by modelling the abstract and concrete visual syntaxt of the Sequence Diagrams formalism in our meta-modelling and model transformation tool AToM^3 [1]. Figure 1 shows the meta-model in AToM^3's Class Diagrams formalism. The latter is similar to UML Class Diagrams. The AToM^3 version allows one to immediately generate a formalism-specific editor, with a generic visual modelling environment, from the Class Diagram and some extra information about visual concrete syntax (shown at the bottom of the figure). The classes and the meaning of their attributes are:

- *Interaction* is a representation of the entire model. All other entities will be contained by this entity, since it is responsible for providing basic UI handling.
- *Lifeline* represents an individual participant in the Interaction. A Lifeline has two attributes which represent the name of the participant class (*instanceName*) and the name of an instance of that class (*className*) respectively.
- *ActionFragment* is used to hold syntactic points (called OccurrenceSpecifications in UML) at the ends of Messages. The ActionFragment is the combination of an atomic ExecutionSpecification and an OccurrenceSpecification of UML 2.0. The main reason for the combination is to simplify the definition of the Graph Grammar rules for transforming Sequence Diagrams to Statecharts (described in Section 3).
- *CombinedFragment* is defined by an interaction operator and corresponding interaction operands. Depending on what kind of operator is defined, specific constraints can be imposed on the number of operands a CombinedFragment can contain. For example, “option (opt)” or “loop” must have exactly one operand.
- *InteractionOperand* is contained in a CombinedFragment. An InteractionOperand, with an optional guard expression, represents one operand of the expression given by the enclosing CombinedFragment.

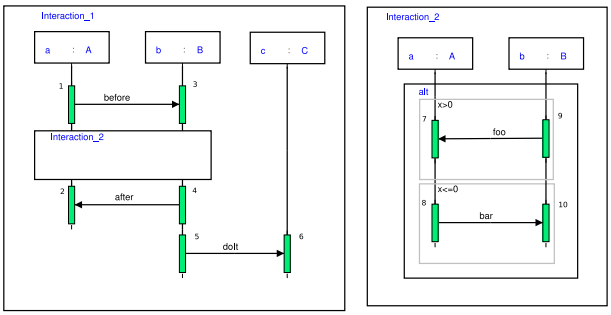

Fig. 2. A simple Sequence Diagrams model in AToM^3

– *InteractionUse* refers to another Interaction. The InteractionUse is a shorthand for copying the contents of the referred Interaction where the InteractionUse is. It is common to want to share portions of an interaction between several other interactions. An InteractionUse allows multiple interactions to reference an interaction that represents a common portion of their specification.

The entities whose icons have a hexagonal shape at the top are generated as relationships/edges. They come in two types, which are set via edit dialogs. The first type is the invisible hierarchical relationship. The following entities are of this type: InteractionContains, CombinedFragmentContains, InteractionOperandContains, and InteractionUseCovers. AToM^3 was extended to internally keep track of such hierarchical relationships, so finding parents and children is easy. The second type of relationship is the visible arrows, which possess attributes just like the nodes did. The visual relationships and the meaning of their attributes are as follows:

– *ActionConnection* represents a connection between a Lifeline and an ActionFragment, or between two ActionFragments which cover the same Lifeline.
– *Message* defines a particular communication between Lifelines of an Interaction. A Message associates two OccurrenceSpecifications - one sending OccurrenceSpecification and one receiving OccurrenceSpecification. The “message” attribute defines the signature of the Message. A message is shown as a line from the sender to the receiver with a filled arrow head. The “isSynchronous” attribute determines whether the message is synchronous of asynchronous.

Figure 2 shows an example Sequence Diagrams model in the visual modelling environment synthesized from the above meta-model. We use this example to illustrate the key concepts and algorithms throughout this paper. On the left hand side of the figure is the interaction $Interaction\_1$ which contains three lifelines $a : A$, $b : B$ and $c : C$, three messages $before$, $after$ and $doIt$, and one interaction use between the two messages which refers to interaction $Interaction\_2$ and covers lifelines $a : A$ and $b : B$. On the right hand side of the figure is the $Interaction\_2$ which contains two lifelines $a : A$ and $b : B$ as $Interaction\_1$ does and one combined fragment whose operator is “alt”. Inside the “alt” combined fragment, there are two alternative operands, “$x > 0$” and “$x <= 0$”, which contain two messages foo and bar respectively. Note that number labels are added for action fragments which are not in the original models for ease of the illustration.

3 Transformation Sequence Diagrams into Statecharts

Transformation of models is a crucial element in all model-based endeavors [2]. As models and meta-models are essentially attributed and typed graphs, we can transform them using graph rewriting. Transformation models are specified in the form of Graph Grammars. Graph grammars are a natural, formal, visual, declarative and high-level representation of the transformation. A Graph Grammar (in AToM^3) is composed of an ordered set of rules. A rule consists of a Left Hand Side (LHS) graph and a Right Hand Side (RHS) graph. Rules can have applicability conditions (pre-conditions) and actions (post-actions) which are checked and performed respectively when the rule is applied.

Rules are evaluated in order against a host graph which represents the model to be transformed. If a graph matching is found between the LHS graph of a rule and a subgraph of the host graph, the rule is eligible to be applied. Then, the pre-condition of the rule is evaluated. If it is true, the rule is applied. When a rule is applied, the matching subgraph of the host graph is replaced by the RHS graph of the rule. After a rule matching and subsequent application, the graph rewriting system starts the search again. The graph grammar execution ends when no more matching rules remain.

Nodes and links in LHSs and RHSs are identified by means of numbers (labels). If a number appears on both the LHS and the RHS of a rule, the node or connection is retained when the rule is applied. If the number appears only on the LHS, the node or connection is deleted when the rule is applied. Finally, if the number appears only on the RHS, the node or connection is created when the rule is applied. Node and connection attributes in LHSs must be provided with attribute values which will be compared with the node and connection attributes of the host graph during the matching process. These attributes can be set to ANY, or may have specific values. In the RHS, we can specify changed attribute values for those nodes which also appear in the LHS. In AToM^3, we can either copy the value of the attributes of the LHS, specify a new value, or associate arbitrary Python code to compute the attribute value, possibly based on other nodes' attributes.

During formalism transformation we have a model in a source formalism that is transformed to a model in a target formalism. The model elements in the source formalism could be related to each other. The application of a rule may introduce the counterpart model element from the target formalism for a model element in the source formalism. Removing the source formalism model elements at this stage will destroy all its relationships and hence we have no way to find out what it connects to. To precisely keep track of source formalism elements and their counterpart elements in the target formalism, we use a GenericGraph formalism which acts as a "helper" during the transformation. The GenericGraph formalism consists of only two types of elements, GenericGraphNode and GenericGraphEdge.

3.1 Sequence Diagrams Model to Statecharts Model

The Sequence Diagrams formalism is used to show the interactions between objects by means of messages arranged in in a (partially) time-ordered sequence. One of the uses of sequence diagrams is to, during the requirements phase of a project, transition from requirements expressed as use cases to the next and more detailed level of refinement.

Statecharts are a core formalism for describing behavior. By executing (or simulating) the model, we can learn the behavior of the system precisely so the requirements of the system can be validated. The transition from interaction diagrams (e.g., Sequence Diagrams, Live Sequence Charts [3] and Use Case Charts [4]) to Finite State Machines (e.g., Statecharts) which satisfy the constraints specified in the interaction diagrams has become one of the key activities in object-oriented analysis and design.

Since the release of the UML 2.0 [6] specification, the Sequence Diagrams formalism has become more powerful and rigorous as it provides more well-defined constructs than before. For example, structured control constructs (Combined Fragments) such as loops, conditionals, and parallel execution, are introduced, which provide a precise way to model complex flow of control. The introduction of Interaction Uses –a reference to another interaction, which is usually defined in its own sequence diagram– provides a natural way to re-use existing sequence diagrams and decompose more complex sequences into simpler ones.

The richness of constructs of the Sequence Diagrams formalism is definitely one advantage, but also a big challenge for its transformation to other formalisms such as Statecharts. For example, since combined fragments are actually nested structures which can form a series of nested scopes inside an interaction, the transformation procedure needs to be able to transform recursively inside each scope and combine results with outer scope ones. This is not an easy nor intuitive task. Another challenge is how to transform interaction uses, i.e., replacing them with actual interactions and combining them for ultimate transformation to other formalisms. The following will explain the strategy and algorithms for tackling these problems in details.

3.2 Four Phase Transformation

We now present the Graph Grammar rules used to transform a Sequence Diagram model to a Statechart model. The graph grammar consists of forty-six rules. In order to explain them clearly, these rules are grouped into four categories according to their function and different phases in which they are executed during a transformation. These four phases will be explained in the subsequent sections. They are summarized here.

1. *Initialization*. In this phase, the source model is parsed, and some global and local data structures used in the following phases to help to control the transformation flow are initialized.
2. *Importation*. In this phase, all Interaction Use fragments are dereferenced, i.e., replaced by the actual Interaction referred to. This is an optional phase, since an interaction is valid without any interaction use. As an interaction use can refer to an interaction which in turn contains other interaction uses, this process can be recursive. However, an interaction can not contain an interaction use which refers to the interaction itself to avoid infinite looping. This constraint (specified in the metamodel of the Sequence Diagrams formalism) is checked when a model is saved. Because importation will change the original model by adding new elements, *Initialization* is needed again after this phase.
3. *Transformation*. This is the core of the transformation. The transformation proceeds in two dimensions. The horizontal dimension follows the order of appearance of

different objects. The vertical dimension follows the order of occurrence of a series (flow) of messages. The procedures ensure the processing is in the correct order in the presence of nested control constructs.

4. *Optimization*. After the previous two phases, a Statecharts model is generated. However, a number of redundant elements were added to the target model in those phases. These are now removed.

3.3 Transformation Illustration by Example

In this section, we use the example shown in Figure 2 to illustrate the key concepts and algorithms of the model transformation.

Phase One: Initialization This phase aims to initialize some data structures (others are initialized in other phases when necessary) used in the next phases to help the control of the transformation flow.

We do not give the complete list of the initialized data structures (or variables) in this paper. In summary, there are two categories: *Global* and *Local*. The global ones are linked with the model graph under transformation and can be accessed anytime and anywhere. For example, *maxScope* keeps the information about the maximum number of scopes an Interaction model currently has, which is determined by how many levels of nested structured control constructs (e.g., Combined Fragments) the model contains. The local ones are linked to specific graph elements which normally can only be accessed when these elements appear in a rule. For example, the *scope* of a Combined Fragment element remembers the number of the scope the element itself is at. For instance, if a Combined Fragment element CF_1 is directly contained inside an Interaction, then $CF_1.scope = 0$; and if another Combined Fragment CF_2 is inside CF_1, then $CF_2.scope = 1$. One important use of this *scope* is to determine whether the element of a matched subgraph is in the right scope the transformation is currently working on. This is one of the pre-conditions to be evaluated to determine if a rule can be executed.

The main procedure of the initialization in the post-action has three steps. We refer to [7] for the details.

1. *setPosition(graph)*. Set the vertical position of each of the Action Fragment and Combined Fragment elements along a Lifeline.
2. *setScope(graph)*. Set the scope of each of the Action Fragment and Combined Fragment elements.
3. *initiateGlobals*. Set default values of other unset global variables.

Phase Two: Importation In this phase, all Interaction Use fragments are replaced by the actual referred to Interactions. This is an optional phase, since an interaction is valid without containing any interaction use. As an interaction use can refer to an interaction which in turn contains other interaction uses, this process can be recursive. Because the importation will modify the original model by adding new elements, *Phase One* is repeated after this phase.

To combine the two interactions, two things need to be determined for each of the lifelines in the imported interaction. First, which action fragment is in the head position along a lifeline, and which one is the tail (they could be the same one). Second, which action fragment is in the position just before the interaction use along the lifeline in the original interaction, and which one is just after that interaction use (there could be none) .

It would be hard to achieve this task in a simple way if we only use the graph matching of graph grammars. So, we use some auxiliary data structures which are not shown in this paper. The procedure of the task is used in the post-action of rule *importInteraction*. Seee [7] for details of the algorithm.

The list of rules, their execution order and short descriptions are given in Table 1. Note that some rules have the same order which can have two meanings. First, it means that there is a mutually exclusive choice between conditions of these rules. For example, only one of *connectHeadAFFirst* and *connectHeadAFFirstPrime* will be executed in one process iteration. If this is not the case, the execution order of these rules is irrelevant and is determined randomly. For example, the rules with order of 8. Rule 2 *connectHeadAFFirst* is shown in Figure 3.

Table 1. Graph Grammar rules in execution order in Phase Two

Order	Rule Name	Description
1	importInteraction	Import the Interaction model referred to by a current target Interaction Use and initialize helper data structures used in the next steps.
2	connectHeadAFFirst	Connect the head ActionFragment of a Lifeline in the imported Interaction with the proper one of the same Lifeline (with the same class name and instance name) but in the original Interaction.
2	connectHeadAFFirstPrime	Connect the head ActionFragment of a Lifeline in the imported Interaction directly with the same Lifeline but in the original Interaction. The checking specified in the meta-model is done when the model is saved.
3	connectHeadAFSecond	Disconnect the ActionFragment of the Lifeline in the original Interaction affected by previous rules from its old successor.
4	connectTailAFFirst	Remove the Lifeline from the imported Interaction as there is no ActionFragment of the same Lifeline in the original Interaction need to be connected.
5	connectTailAFSecond	Connect the tail ActionFragment of a Lifeline in the imported Interaction with the proper one of the same Lifeline (with the same class name and instance name) but in the original Interaction.
6	connectTailAFThird	Disconnect the ActionFragment of the Lifeline in the original Interaction affected by previous rules from its old predecessor.
7	createWrapperCF	Create a CombinedFragment (together with an InteractionOperand inside) for holding all ActionFragments, CombinedFragments and InteractionUses inside the imported Interaction in order to move them into the original Interaction in one time.
8	connectAFWithWrapperCF	Wrap ActionFragments inside the CombinedFragment.
8	connectCFWithWrapperCF	Wrap CombinedFragments inside the CombinedFragment.
8	connectIUWithWrapperCF	Wrap InteractionUses inside the CombinedFragment.
9	cleanImportedLifeline	Remove Lifeline of the imported Interaction.
10	cleanImportedInteraction	Connect the wrapper CombinedFragment created earlier with the original Interaction, then remove the imported one and the Interaction Use that refers to it.

The result of the transformation on the example sequence diagram is shown in Figure 4. Note that number labels are added for action fragments which are not in the original models for illustration. Note how a special “seq” combined fragment which has only one operand is added by rule *createWrapperCF* to enclose all elements im-

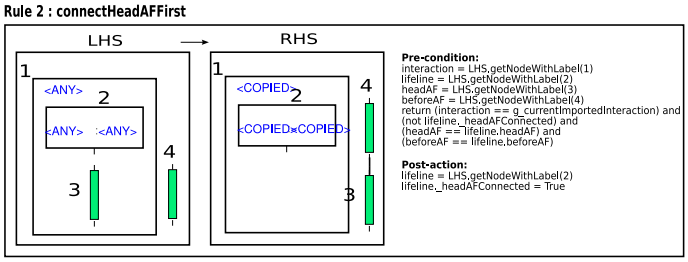

Fig. 3. Graph Grammar rule *connectHeadAFFirst* in Phase Two

ported from *Interaction_2*. It is redundant after this phase and will be removed by the optimization described in Section 3.3.

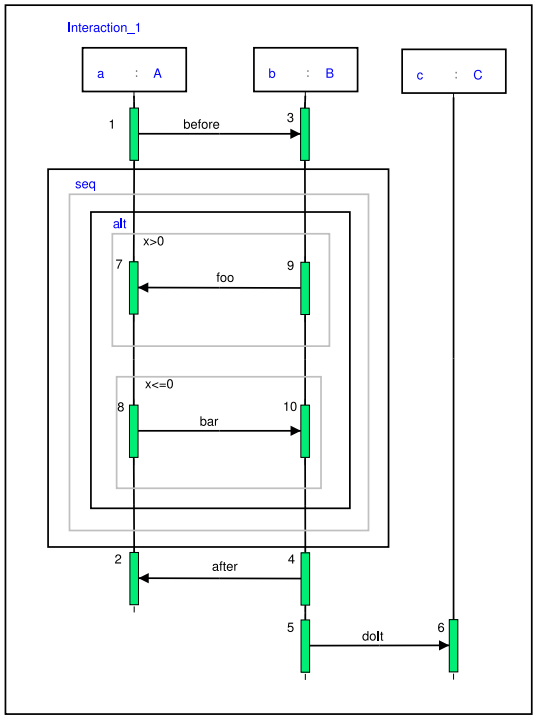

Fig. 4. Transformation result of Phase Two

As an interaction may contain more than one interaction use, the strategy to control the flow of the transformation is to ensure we transform one interaction use at a time. That is, one iteration of the set of rules in this phase corresponds to the transformation of one interaction use. This is implemented by evaluating and manipulating the global variable *g_currentInteractionUse*.

Phase Three: Transformation This is the core phase which maps Sequence Diagrams to Statecharts. The transformation proceeds in two dimensions. The horizontal dimension follows the order of appearance of different lifelines. The vertical dimension fol-

lows the order of occurrences of a series of messages. The procedures ensure both dimensions are processed in correct order with respect to nested control constructs.

The strategy is depicted in Figure 5. Each iteration starts with finding an unprocessed lifeline element. It then proceeds with messages and fragments along the lifeline from top to bottom. If the interaction contains nested fragments, the process continues from the outermost to the innermost of nested scopes.

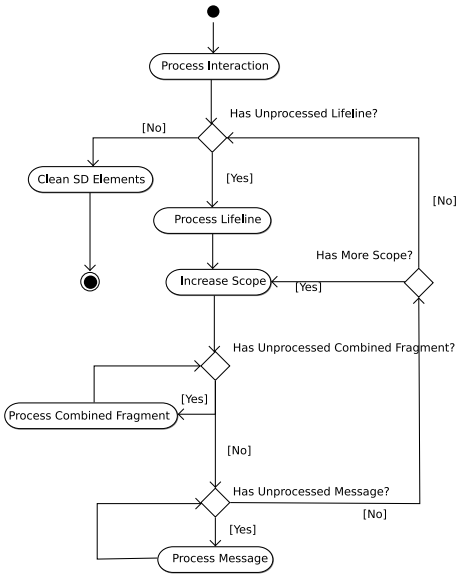

Fig. 5. The strategy of Phase Three

The list of rules, their execution order and short descriptions are given in Table 2. Rule 13 *combinedFragment* is shown in Figure 6.

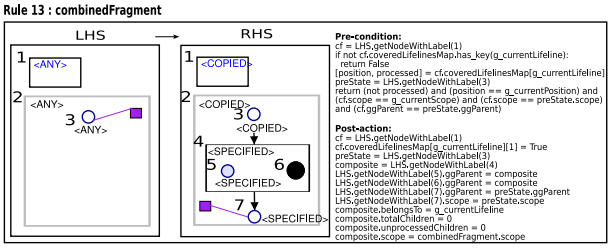

Fig. 6. Graph Grammar rule *combinedFragment* in Phase Three

Rule *msgIn* states that a message directed towards a lifeline becomes a triggering event in the Statecharts. Rule *msgOut* states that a message directed away from a lifeline becomes an action to send that message in the Statecharts. Both of these rules result in a transition to a new state. A combined fragment results in a transition to a new composite state which is a container of enclosed orthogonal compo-

Table 2. Graph Grammar rules in execution order in Phase Three

Order	Rule Name	Description
11	interaction	Transform an Interaction element to a DChart element, with two new state nodes and one new composite node enclosed and a link to the original Interaction element.
12	lifeline	Transform a Lifeline node to an Orthogonal node, with two new state nodes enclosed and a link to the original Lifeline node.
13	combinedFragment	Transform a CombinedFragment element, which is in the current transforming level, to a composite node, with two new state nodes enclosed and one new state node concatenated and a link to the original CombinedFragment element.
14	interactionOperand	Transform an InteractionOperand element, enclosed in a transformed CombinedFragment element, to an new Orthogonal element enclosed in a new composite element, with two new state nodes enclosed.
15	operatorParallel	Adjust a transformed "parallel" CombinedFragment.
15	operatorLoop	Adjust a transformed "loop" CombinedFragment.
15	operatorOption	Adjust a transformed "option" CombinedFragment.
18	msgIn	Transform an incoming event (Message) along the current Lifeline, to a transition and a state node.
18	msgOut	Transform an outgoing event (Message) along the current Lifeline, to a transition and a state node.
19	setScope	Increase the scope to start the transformation of another nested level. No visual syntax.
20	connectWithFinal	Connect the last Basic node in any enclosing fragment with a "final" state node.
21	cleanLifeline	Remove Lifelines.
22	cleanActionFragment	Remove Lifelines.
23	cleanInteraction	Remove Interaction.
24	cleanCombinedFramgment	Remove CombinedFramgments.
25	cleanInteractionOperand	Remove InteractionOperands.

nents and states transformed from interaction operands and messages by the following rules. Rule *interactionOperand* is a general rule for all kinds of combined fragment operators. Based on the result of that rule, more operations are implemented in rule *operatorParallel*, *operatorLoop* and *operatorOption* for some specific operators. For example, different orthogonal components representing branches of a "parallel" combined fragment are combined into one composite state by rule *operatorParallel*; an extra transition from "final" state to "default" state of a "loop" combined fragment is added by rule *operatorLoop*; an extra transition with the opposite guard condition from "final" state to "default" state of a "option" combined fragment is added by rule *operatorOption*. It is easy to find that some constructs are redundant after the transformation of this phase. For example, for "loop" combined fragment, the corresponding composite state has no need to contain its own "default" and "final" states and another enclosed composite state (and even its enclosed orthogonal component), if the "guard" information can be somehow moved to somewhere at the inside and the loop-back transition can also be connected between the "default" and "final" states of the inside. We consider all of such situations and optimize these structures as much as possible in the next phase, Optimization, to make the generated Statecharts more readable for humans and more efficient for simulation and analysis. The result of the transformation upon the example Sequence Diagrams is shown in Figure 7. There are many states and many levels of nested composite components. Although this is a correct synthesized statechart, most states and components are redundant and can be optimized away.

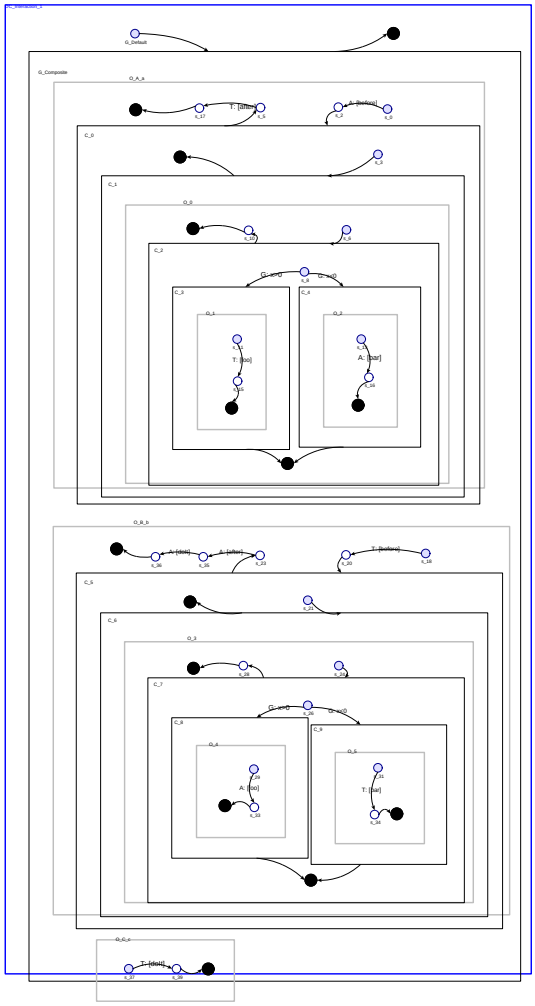

Fig. 7. Transformation result of Phase Three

Phase Four: Optimization After the previous two phases, an executable Statecharts model is generated. However, as discussed earlier, a number of redundant elements are also generated in the target model in those phases, which makes it hard to read and refine. Now it is time to make some optimization to get the final compact, readable and efficient Statecharts model. This phase is analogous to the optimization phase after the code generation of a compilation.

There are three subphases for the optimization:

1. *Orthogonal optimization*. Any orthogonal component which is the only enclosed orthogonal of the parent composite, is considered redundant and is removed. All states and composites enclosed by the removed orthogonal component become children of the parent composite.
2. *Composite optimization*. Any composite which does not enclose any orthogonal component is considered redundant and is removed. Again, all enclosed states and

composites become children of the parent component. The enclosed “default” and “final” state are replaced by normal states and connected with outer states by transitions. As a parent component could be a composite or an orthogonal component, there are two sets of rules for this subphase. Since the composites could be nested, this recursive process starts from the outermost one.

3. *Transition and state optimization*. Any transition without triggering and action and its guard always true, is redundant and is removed.

Rule 31 *compositeOptThird* and Rule *compositeOptFour* are shown in Figure 8. The list of rules, their execution order and short descriptions are given in Table 3.

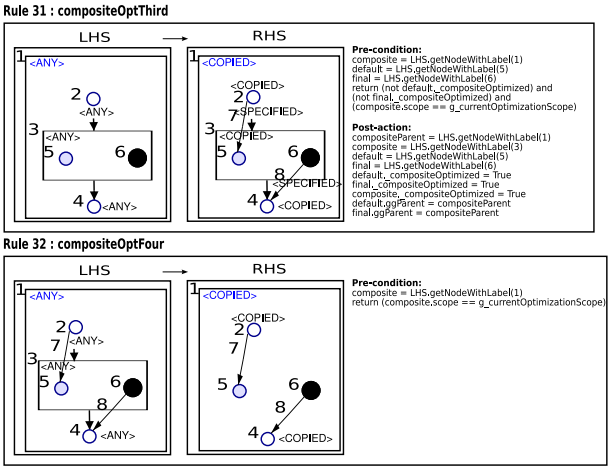

Fig. 8. Graph Grammar rules *compositeOptThird* and *compositeOptFour* in Phase Four

After these optimizations, the final statecharts transformed from the example of Figure 2 is shown in Figure 9.

4 Requirements Document Generation from Sequence Diagrams

Before requirements are captured in the form of a series of Sequence Diagrams, they are usually expressed in the form of Use Cases (in plain text or visually) by domain experts or end users. In this case, it is useful to automate the mapping from a Use Case to a Sequence Diagram. Li [8] proposed a semi-automatic approach to translate a use case to message sends. However, in order to reduce the complexity of parsing a natural language and the vagueness, the approach requires that requirements be normalized before translation. Our approach to briding the gap between Use Cases and Sequence Diagrams is the inverse of the aforementioned mapping. As mentioned previously, our model-driven approach supports that well-defined Sequence Diagrams can be used to generate textual representation of requirements in a natural language the customers are familiar with. The generated requirements can be used for evaluation and immediate feedback. This process is fully automatic and provides a useful way to refine requirements at an early stage in the development process.

Table 3. Graph Grammar rules in execution order in Phase Four

Order	Rule Name	Description
26	orthogonalOptFirst	Reconnect the enclosed state node of the removing orthogonal component with its parent composite.
27	orthogonalOptSecond	Reconnect the enclosed composite node of the removing orthogonal component with its parent composite.
28	orthogonalOptThird	Remove the orthogonal component which is the only enclosed orthogonal component of the parent composite.
29	compositeOptFirst	Reconnect the enclosed state node of the removing composite with its parent composite.
30	compositeOptSecond	Reconnect the enclosed composite node of the removing composite with its parent composite.
31	compositeOptThird	Copy transitions originally to and from the removing composite to new transitions of its inside states.
32	compositeOptFour	Remove the composite which does not enclose any orthogonal component.
33	setOptimizationScope	Increase the scope of nesting for optimization to starts another iteration. No visual syntax.
34	compositeOptFirstPrime	Reconnect the enclosed state node of the removing composite with its parent orthogonal component.
35	compositeOptSecondPrime	Reconnect the enclosed composite node of the removing composite with its parent orthogonal component.
36	compositeOptThirdPrime	Copy transitions originally to and from the removing composite to new transitions of its inside states.
37	compositeOptFourPrime	Remove the composite which does not enclose any orthogonal component.
38	cleanStateFirst	Remove state whose only outgoing transition is redundant which has one incoming transition.
38	cleanStateSecond	Remove state whose only outgoing transition is redundant which has two incoming transitions.
39	finalstateOptFirst	Replace each incoming transition of a final state (except for the one in the top level) by a copied transition to a new state.
40	finalstateOptSecond	Remove final states (except for the one in the top level).

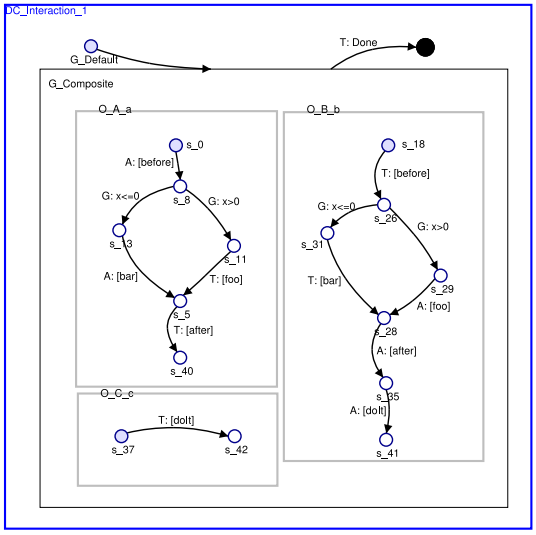

Fig. 9. The final generated Statechart for the example of Figure 2

The generated text description of a message send follows the simple format:

$$sourceactor + \text{“}sendsmessage\text{”} + message + \text{“}to\text{”} + targetactor$$

if there is no customized description of the message, i.e., the “description” attribute of the message is left empty. Otherwise, if a user gives a more natural description to a

message, the generated text description of the message send follows this format:

$$sourceactor + customizeddescription + targetactor$$

As one can see, the above format is insufficient to deal with the case when there is a need that either the "source actor" or "target actor" should appear in the middle of a description. In order to make the generated text more natural to read, we provide two macro variables, "$SOURCE$" and "$TARGET$", which can be included anywhere in a customized description. They represent the real "source actor" and "target actor" respectively. That is, the generation algorithm will replace a macro variable with the real actor. "source actor" or "target actor" comes from its lifeline's "instanceName" or "className" attribute. If "instanceName" is not empty then "instanceName" is used to represent the actor; otherwise, "className" is used. The generated text description of an interaction use follows the simple format:

$$\text{"}Interaction\text{"} + refersTo + \text{"}isimportedhere\text{"}$$

We apply our simple algorithm (given in detail in [7]) to the simple Sequence Diagrams model (shown in Figure 2) to generate textual requirements. The result is shown below. Note that for this example, messages of Sequence Diagrams models *Interaction_1* and *Interaction_2* are all annotated with customized descriptions. Some of these customized descriptions contains macro variable "$TARGET$" in the middle of the text. For example, in *Interaction_1*, the annotated description of message send 1 is "asks $TARGET$ to start a service", in which "$TARGET$" is then translated into "b".

```
Use Case: Interaction_1
Actors: a, b, c
Scenario:
 1. a asks b to start a service
 2. Interaction 'Interaction_2' is imported here
 3. b reports status to a
 4. b asks c to log the status

Use Case: Interaction_2
Actors: a, b
Scenario:
 1. If x>0:
 2.   b returns a service handler to a
 3. Else if x<=0:
 4.   a asks b to start another service
```

5 Related work and Conclusions

We proposed a model-driven approach to transform software requirements. The model-driven approach starts with modelling requirements of a system in Sequence Diagrams, and the subsequent automatic transformation back into textual requirements or into Statecharts. A visual modelling environment was built in AToM^3 using *Meta-modelling* and *Model Transformation* to support the model-driven approach. It supports modelling in Sequence Diagrams, automatic transformation to Statecharts, and automatic generation of requirements text from Sequence Diagrams.

Sutcliffe et. al. [9] proposed a method and a tool for specification of use cases, automatic generation of scenarios from use cases and semi-automatic validation based-on

generated scenarios. Harel et. al. [10, 11] proposed a play-in/play-out approach to capture behavioral requirements. The Play-Engine automatically constructs corresponding requirements in the scenario-based language of Live Sequence Charts (LSCs) [3], and finally semi-automatically synthesizing a collection of finite state machines. Whittle et al. propose use case charts in [4] which is a 3-level notation based on extended activity diagrams used for specifying use cases. In [12], algorithms are presented that transform use case charts into a set of hierarchical state machines which then can be used for simulation, test generation and validation. The algorithm starts with the conversion to hierarchical state machines for a level-3 sequence diagram, then proceeds with the combination of a set of hierarchical state machines generated for each scenario node at level-2, and finally completes the synthesis of the final hierarchical state machine for the level-1 use case chart by further combining generated hierarchical state machines. The core of the algorithm is the synthesis of hierarchical state machines from a sequence diagram. Our approach is very similar to that of Whittle and was developed concurrently. We have chosen to use the full power of sequence diagrams which alleviates the need for a 3-level notation. We plan to provide support for Whittle's process and notation in the near future.

References

1. de Lara, J., Vangheluwe, H., Alfonseca, M.: Meta-modelling and graph grammars for multi-paradigm modelling in AToM^3. Software and System Modeling **3**(3) (2004) 194–209
2. Vangheluwe, H., de Lara, J.: Computer automated multi-paradigm modelling for analysis and design of traffic networks. In: Winter Simulation Conference. (2004) 249–258
3. Damm, W., Harel, D.: Lscs: Breathing life into message sequence charts. Formal Methods in System Design **19**(1) (2001) 45–80
4. Whittle, J.: Specifying precise use cases with use case charts. In: MoDELS Satellite Events. (2005) 290–301
5. Rhapsody. I-Logix, Inc., http://www.ilogix.com/products/
6. Modeling Language 2.0 Specification, U., http://www.omg.org. (2005)
7. Ximeng Sun: A model-driven approach to scenario-based requirements engineering. Master's thesis, McGill University (2007)
8. Li, L.: Translating use cases to sequence diagrams. In: ASE. (2000) 293–296
9. Sutcliffe, A.G., Maiden, N.A., Minocha, S., Manuel, D.: Supporting scenario-based requirements engineering. Software Engineering **24**(12) (1998) 1072–1088
10. David Harel and Rami Marelly: Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine. Springer-Verlag (2003)
11. Harel, D., Kugler, H., Pnueli, A.: Synthesis revisited: Generating statechart models from scenario-based requirements. In: Formal Methods in Thanks a lot! Software and Systems Modeling. (2005) 309–324
12. Whittle, J., Jayaraman, P.K.: Generating hierarchical state machines from use case charts. Proceedings of the 14th IEEE International Requirements Engineering Conference (RE'06) **0** (2006) 16–25