A Model-Driven Approach to Scenario-Based Requirements Engineering

Ximeng Sun
Supervisor : Prof. Hans Vangheluwe

School of Computer Science
McGill University, Montréal, Canada

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfilment of the requirements of the degree of
Master of Science in Computer Science

Abstract

A model-driven approach to scenario-based requirements engineering is proposed. The approach, which is based on Computer Automated Multi-Paradigm Modeling (CAMPaM), aims to improve the software process. A framework is given and implemented to reason about models of systems at multiple levels of abstraction, to transform between models in different formalisms, and to provide and evolve modeling formalisms.

The model-driven approach starts with modeling requirements of a system in scenario models and the subsequent automatic transformation to state-based behavior models. Then, either code can be synthesized or models can be further transformed into models with additional information such as explicit timing information or interactions between components. These models, together with the inputs (e.g., queries, performance metrics, test cases, etc.) generated directly from the scenario models, can be used for a variety of purposes, such as verification, analysis, simulation, animation and so on.

A visual modeling environment is built in AToM^3 using *Meta-Modeling* and *Model Transformation*. It supports modeling in Sequence Diagrams, automatic transformation to Statecharts, and automatic generation of requirements text from Sequence Diagrams.

An application of the model-driven approach to the assessment of use cases for dependable systems is shown.

Résumé

Nous proposons ici une approche dirigée par des modèles et basée sur des scénarios. Cette approche, basée sur les CAMPaM (Computer Automated Multi-Paradigm Modeling), a pour but d'améliorer les processus systèmes. Un espace de travail est alloué et implementé pour définir les modèles de systèmes ayant plusieurs niveaux d'abstraction. Il est aussi possible de transformer un modèle en un autre à l'aide de différents formalisnes, de les concevoir, ou de les faire évoluer.

L'approche dirigée par des modèles commence par la modélisation des conditions du système en différents scénario suivant une transformation automatique des comportements des modèles de base. Ensuite, le code et les modèles peuvent être réciproquement synthetisé et tranformé en modèle contenant plus d'informations (temps, interactions entre les objets etc.). Ces modèles, couples avec les informations d'entrée générées par les modèles de scénario(e.g., requètes), peuvent être utilisés pour de multiples objectifs, tel que l'analyse, l'animation, la simulation ou encore la vérification ...

AToM^3 (logiciel de *Meta-Modeling*, et de *Model Transformation*) utilise une interface graphique pour la modélisation. il est possible de modéliser des Sequence Diagrams, de transformer automatiquement ces modèles en Statecharts (diagrammes d'états), et de génération automatique de texte de conditions des Sequence Diagrams.

Une application des approches dirigées par des modèles évaluant des cas d'utilisations pour systèmes est ici detaillée.

Acknowledgements

First of all, I would like to thank my supervisor, Prof. Hans Vangheluwe. He has earnestly supervised my research from the very beginning with his insight and passion for modeling, simulation and software design.

I sincerely thank my colleagues in MSDL (Modeling, Simulation, and Design Lab) Denis Dubé, Sagar Sen, Miriam Zia, Ernesto Posse, Paulet Thierry and Gardette Julien for their selfless help with regard to the tool support, ideas for my thesis, and French.

A special thanks to Sadaf Mustafiz. Our previous work on the model-driven assessment of use cases for dependable systems is the base of the case study of this thesis.

Finally, a big thank you to my mom and my girlfriend for their support in the past, present, and the future in all my endeavors.

Contents

List of Figures

List of Tables

List of Algorithms

Introduction

Requirements engineering is concerned with the acquisition, analysis, specification, validation, and management of requirements of the software system under construction.

Scenarios provide an excellent means of communication between software project stakeholders and are an intuitive way of guiding them through the process of requirements engineering. They can play different roles [Sut03] in the following activities [NE00] of the process:

- Representing the real world to produce models during requirements analysis;
- Inspiring system designs and generating testing material;
- Transforming models and requirements specifications into designs and eventually implementations;
- Reasoning and validating designs;
- Communicating requirements among different stakeholders;
- Maintaining agreement with all stakeholders during the elicitation and modeling of requirements;
- Managing requirements changing.

Some benefits [RC95] can be gained by using scenarios, such as, minimizing the gap between specification and implementation, avoiding mismatches between user/engineer view, providing a rich view of goals, actions and experiences of users and getting good concepts for extending and redesigning existing systems.

While scenarios have become an established technique in the requirements engineering process, many questions still remain for further research. Some of them are listed here:

- Keeping consistency among the models used at different phases (or levels) of the process;
- Keeping consistency of the models when they evolve with requirements;
- Reusing requirements models;
- Deciding when the scenario specification is complete;
- Capturing and analyzing non-functional requirements.

Computer Automated Multi-Paradigm Modeling (CAMPaM) [VdL03] aims to simplify the modeling of complex systems by establishing a framework to reason about models of systems at multiple levels of abstraction, transforming between models in different formalisms, and providing and evolving modeling formalisms. Based on the aspects that it addresses, CAMPaM can be naturally applied to the problem of improving scenario-based requirements engineering. For example, modeling, analysis and validation at multiple levels and in different views, can be achieved by model abstraction and multi-formalism modeling in an automatic and consistent way. As another example, rich methods for verification, analysis, simulation and execution of the target system provided by CAMPaM, can maintain the agreement with customers and increase their acceptability and satisfaction.

In this thesis, we propose a model-driven approach to scenario-based requirements engineering based on CAMPaM, and we present the current implementation of the overall approach.

In Chapter 1, we introduce our model-driven approach to scenario-based requirements engineering. In Chapter 2, the implementation of the first step of our approach, which is the meta-modeling of Sequence Diagrams (one of the key scenario-based languages for capturing requirements) is presented. In Chapter 3, a full description of the explicit modeling of model transformations from Sequence Diagrams to Statecharts (one of the state machine-based modeling languages, which can be easily used for requirements simulation, validation and analysis) is given. The automatic generation of requirements text from Sequence Diagrams is also explained. Finally, in Chapter 4, a case study is given in detail demonstrating how our approach can be applied on requirements elicitation.

1

A Model-Driven Approach to Scenario-Based Requirements Engineering

1.1 Scenario-Based Requirements Engineering

A *requirement* is a feature that the system must have or a constraint that it must satisfy to be accepted by the client [BD03]. *Requirements engineering* is concerned with the acquisition, analysis, specification, validation, and management of requirements of the software system under construction. Among all activities of requirements engineering, there are two main activities: *requirements elicitation*, which results in specifications of the system that the customer understands, is to systematically extract and describe the requirements of the system; and *requirements analysis*, which results in analysis models, aims to formalize the requirements specifications produced during requirements elicitation. The resulting analysis models can be used to validate, correct and clarify the requirements with clients and users for requirements refinement. Then, the complete and unambiguous analysis models become the input of the following system development activities, such as high-level system design, validation, testing and so on. For requirements analysis, two types of models are used to give a formal description of the system: static models, which represent knowledge about relationships, such as Class Diagrams and Entity-Relationship Diagrams; and dynamic models, which represent knowledge about behavior, such as Sequence Diagrams and State Diagrams.

Scenarios and use cases are the most important tools used in requirements elicitation. A *scenario* describes an example of system use in terms of a series of interactions between the user and the system. A *use case* is an abstraction that describes a class of scenarios. Scenarios are the starting point for all modelling and design, and contribute to several parts of the design process.

Since scenario-based requirements engineering has been advocated as an effective means of improving the process of requirements engineering, many methodologies and tools [SMMM98, Li00, Dav03, HKP05, WJ06, Whi05] are developed to try to formalize and automate some stages of this process. Sutcliffe et. al. [SMMM98] 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. [Dav03, HKP05] 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) [DH01], and finally semi-automatically synthesizing a collection of finite state machines. Whittle et. al. [WJ06, Whi05] developed a scenario-based language called Use Case Charts (UCCs) to capture scenario-based requirements and presented algorithms for automatic generation of hierarchical state machines from scenario-based models.

Two key activities are apparent in the Harel's and Whittle's approaches: to find a proper scenario-based language, e.g., LSCs or UCCs, to capture and formalize requirements; to transform scenarios into an executable form, i.e., state machines, which can be easily used for requirements simulation, validation and analysis. Both of these are clearly model-driven approaches.

1.2 Scenario-Based Modeling Languages

1.2.1 Sequence Diagrams

Sequence Diagrams are a visual modeling language. They are the UML variant of Message Sequence Charts, used primarily to show the interactions between objects in the sequential order that those interactions occur. One of the primary uses of sequence diagrams is in the transition from requirements expressed as scenarios to the next level of refinement. Scenarios are often refined into one or more sequence diagrams.

The main purpose of a sequence diagram is to define event sequences that result in some desired outcome. The focus is less on the messages themselves and more on the order in which messages occur. Nevertheless, most Sequence Diagrams will communicate what messages are sent between a systems objects as well as the order in which they occur. The diagram conveys this information along the horizontal and vertical dimensions: the vertical dimension shows, top down, the time sequence of messages/calls as they occur, and the horizontal dimension shows, left to right, the object instances that the messages are sent to/from.

1.2.2 Use Case Charts

Use Case Charts, a 3-level notation based on extended activity diagrams, was proposed by Jon Whittle [Whi05], as a way of specifying use cases in detail, in a way that combines the formality of precise modeling with the ease of use of existing notations.

A use case chart, illustrated in Figure 1.1 [Whi05], specifies the scenarios for a system as a 3-level, use case-based description: level-1 is an extended activity diagram where the nodes are use cases; level-2 is a set of extended activity diagrams where the nodes are scenarios; level-3 is a set of UML2.0 interaction diagrams. Each level-1 use case node is defined by a level-2 activity diagram (i.e., a set of connected scenario nodes). This diagram is called a scenario chart. Each level-2 scenario node is defined by a UML2.0 interaction diagram.

By using these three levels of abstraction to model system requirements, Use Case Charts improves requirements modeling and analysis by providing a precise way for relating and grouping scenarios (usually modeled in a set of interaction diagrams).

Whittle and Jayaraman [WJ06] presented algorithms 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 how to synthesize hierarchical state machines from a sequence diagram.

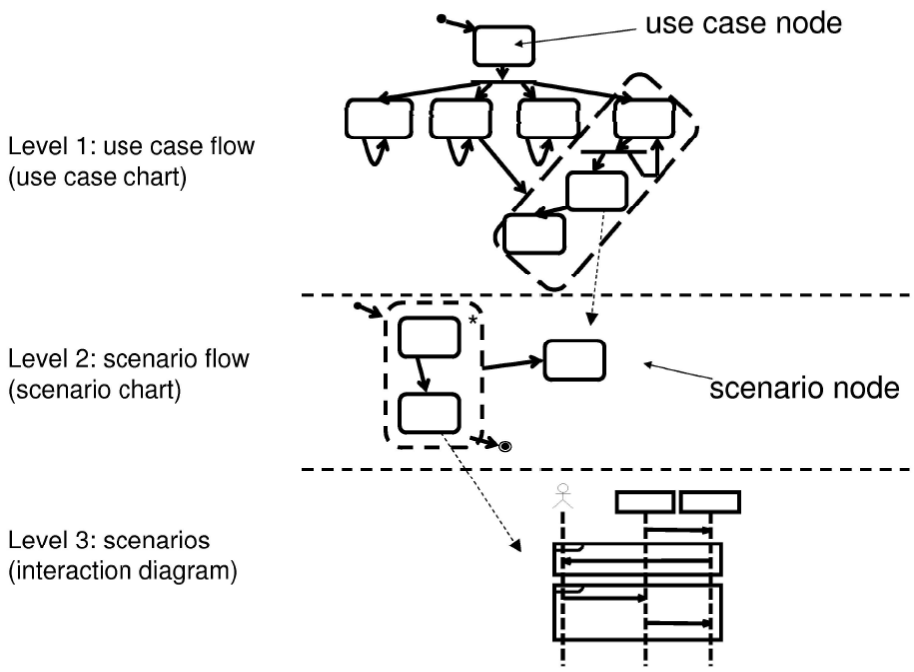

Figure 1.1: Use Case Charts

1.2.3 Statecharts

Statecharts, introduced by David Harel [Har87], are a visual and executable formalism for modeling complex reactive systems. It has roots in the Finite State Automata (FSA) formalism and adds new concepts to it. Those new concepts make the formalism suitable for specifying discrete event systems.

Defining statecharts requires first describing the finite state automata (FSA) formalism it extends. A finite state automaton consists of states and transitions between states. One state is the initial state and this is the first "active" state. Each transition has a trigger, which is a symbol (or event). If a symbol is provided to the FSA and the active state has a transition that has that symbol as a trigger, the target state of the transition then becomes the active state. If no such transition exists, then the FSA halts, since it does not recognize the language, the set of input symbols, that were fed to it. The FSA also includes at least one "final" state. If a transition sets the active state to be one of the final states and the input is at an end, then the FSA has accepted the input sequence of symbols.

Although the FSA formalism is powerful enough to define regular expressions, it is inadequate to express conditional transitions, actions, hierarchy, and concurrency. Fortunately, David Harel extended FSA to deal with these in [Har87]. Conditional transitions are simply transitions that will fire on an event only if both the trigger is matched and the conditional statement evaluates to true. Actions are events or code that is executed whenever a transition fires, a state is entered, or a state is exited. Hierarchy is achieved by adding composite states that can contain other states, including more composite states. The inside of a composite state is a FSA in its own right, with its own default state (the equivalent of an initial FSA state). Since transitions can exit and return to a given hierarchical level, it becomes necessary to add

history states as well. These history states restore the active state within the composite state when a transition returns to the composite state. Hence history allows overriding the default state. Finally, concurrency is added using orthogonal partitions of a composite state. Each orthogonal partition is a simultaneously executing FSA, each with an active state.

1.2.4 DCharts

DCharts, a formalism created by Thomas Feng [Hui04], is a combination of some features of the DEVS formalism (Discrete EVent Systems specification) and Statecharts. Both DEVS and Statecharts can be mapped to DCharts, so in terms of expressive power DCharts is at least as powerful as these two. Denis Dubé has rebuilt a visual modeling environment for the DCharts formalism by applying his formalism-specific user-interface and layout techniques [Den06].

1.3 Model-Driven Approaches

1.3.1 Modeling

Models are an *abstraction* of reality. The structure and behavior of systems we wish to analyze or design can be represented by models. These models, at various *levels of abstraction*, are always described in some *formalism* or *modeling language*. In addition to the *syntax* of a model (how it is represented), one needs to also specify its *meaning* (*i.e.,* assign *semantics*).

One can for example specify on the one hand how a system dynamically evolves over time. On the other hand, it is possible to concentrate purely on the static structure of the system, without specifying its dynamic transitions between states. This demonstrates how, depending on the circumstances, one has to choose the right modeling abstraction.

As another example, during the analysis phase of a software project, models are typically very informal and at a high level of abstraction ("back of the envelope sketches"). During detailed design on the other hand, models are meticulously specified to enable, for example, code generation for embedded systems.

In many cases, a combination of multiple models representing different views, at various levels of abstraction, and using a plethora of formalisms, is required to fully describe the system under study.

1.3.2 Dissecting a Modeling Language

To "model" modeling languages and ultimately synthesize visual modeling environments for those languages, we will break down a modeling language into its basic constituents. This is illustrated in Figure 1.2. It is inspired by the description by Harel and Rumpe [HR00].

As mentioned previously, the two main aspects of a model are its syntax (how it is represented) on the one hand and its semantics (what it means) on the other hand.

The syntax of modeling languages is traditionally partitioned into *concrete syntax* and *abstract syntax*. In textual languages for example, the concrete syntax is made up of sequences of *characters* taken from an *alphabet*. These characters are typically grouped into *words* or *tokens*. Certain sequences of words or *sentences* are considered valid (*i.e.,* belong to the language). The (possibly infinite) *set* of all valid sentences is said to make up the language. Costagliola et. al. [CLOP02] present a framework of visual language classes in which the analogy between textual and visual characters, words, and sentences becomes apparent. Visual languages are those languages whose concrete syntax is visual (graphical, geometrical, topological, ...) as opposed to textual.

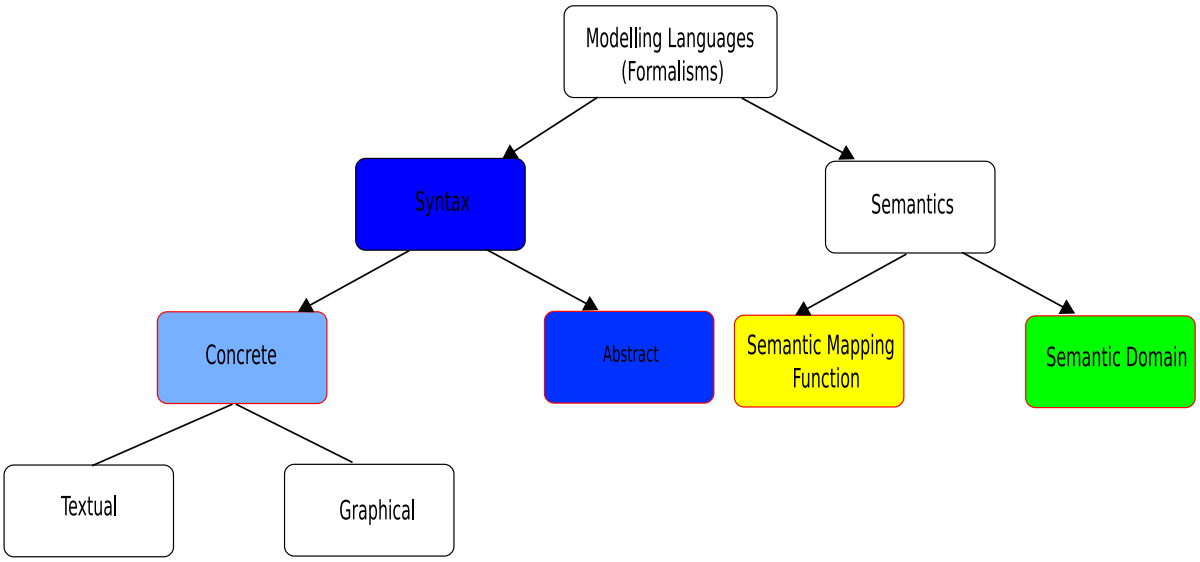

Figure 1.2: Modeling Language Breakdown

For practical reasons, models are often stripped of irrelevant concrete syntax information during syntax checking. This results in an "abstract" representation which captures the "essence" of the model. This is called the *abstract syntax*. Obviously, a single abstract syntax may be represented using multiple concrete syntaxes. In programming language compilers, abstract syntax of models (due to the nature of programs) is typically represented in *Abstract Syntax Trees* (ASTs). In the context of general modeling, where models are often graph-like, this representation can be generalized to *Abstract Syntax Graphs* (ASGs).

Once the syntactic correctness of a model has been established, its meaning must be specified. This meaning must be *unique* and *precise*. Meaning can be expressed by specifying a *semantic mapping function* which maps every model in a language onto an element in a *semantic domain*. For example, the meaning of a Causal Block Diagram is given by mapping it onto an Ordinary Differential Equation. For practical reasons, semantic mapping is usually applied to the abstract rather than to the concrete syntax of a model. Note that the semantic domain is a modeling language in its own right which needs to be properly modeled (and so on, recursively). In practice, the semantic mapping function maps abstract syntax onto abstract syntax.

To continue the introduction of meta-modeling and model transformation concepts, languages will explicitly be represented as (possibly infinite) sets as shown in Figure 1.3. In the figure, insideness denotes the sub-set relationship.

The dots represent model which are elements of the encompassing set(s).

As one can always, at some level of abstraction, represent a model as a graph structure, all models are shown as elements of the set of all graphs Graph. Though this restriction is not necessary, it is commonly used as it allows for the design, implementation and bootstrapping of (meta-)modeling environments. As such, any modeling language becomes a (possibly infinite) set of graphs. In the bottom center of Figure 1.3 is the abstract syntax set A. It is a set of models stripped of their concrete syntax.

Meta-modeling is a heavily over-used term. Here, we will use it to denote the explicit description (in the form of a model in an appropriate meta-modeling language) of the abstract syntax set

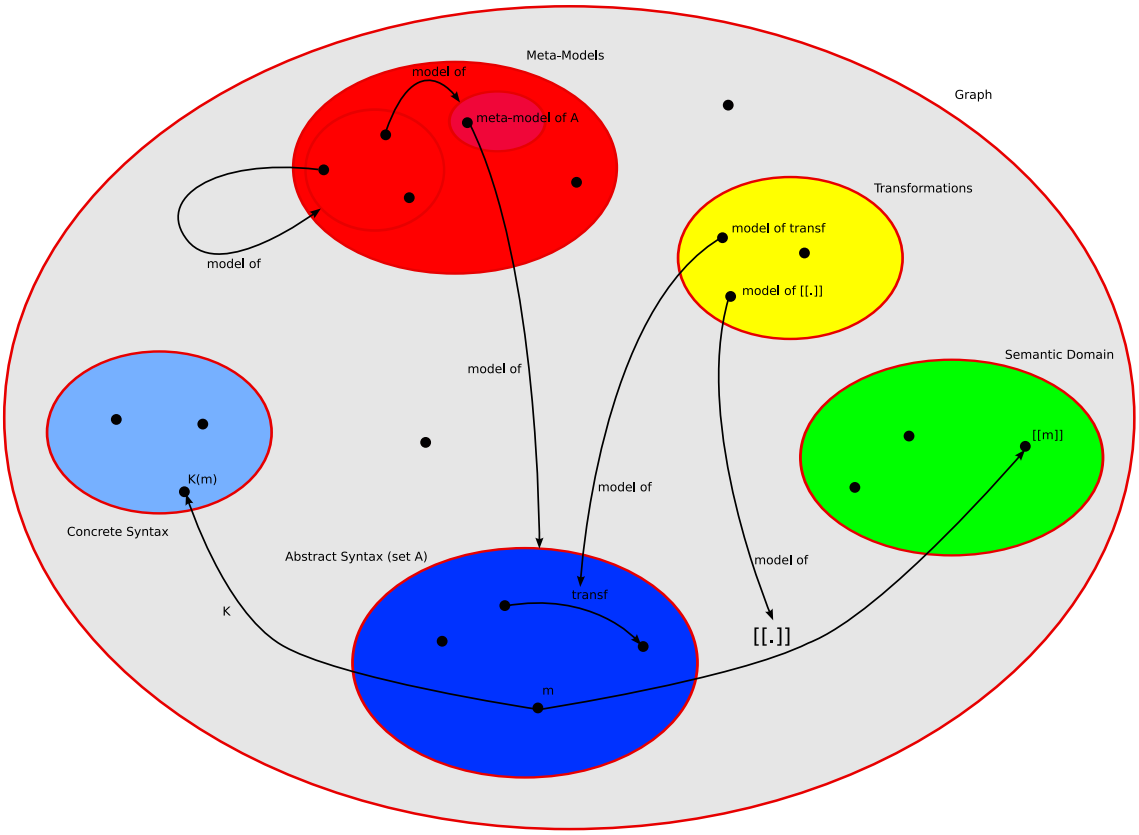

Figure 1.3: Modeling Languages as Sets

A. Often, meta-modeling also covers a model of the concrete syntax. Semantics is however not covered. In the figure, the set A is described by means of the model meta-model of A. On the one hand, a meta-model can be used to *check* whether a general model (a graph) *belongs to* the set A. On the other hand, one could, at least in principle, use a meta-model to *generate* all elements of A. This explains why the term meta-model and grammar are often used inter-changeably.

Several languages are suitable to describe meta-models in. Two approaches are in common use:

1. A meta-model is a *type-graph*. Elements of the language described by the meta-model are instance graphs. There must be a *morphism* between an instance-graph (model) and a type-graph (meta-model) for the model to be in the language. Commonly used meta-modeling languages are Entity Relationship Diagrams (ERDs) and Class Diagrams (adding inheritance to ERDs). The expressive power of this approach is often not sufficient and an extra *constraint language* (such as the Object Constraint Language in the UML) specifying constraints over instances is used to further specify the set of models in a language. This is the approach used by the OMG to specify the abstract syntax of the Unified Modeling Language (UML).
2. A more general approach specifies a meta-model as a transformation (in an appropriate formalism such as Graph Grammars) which, when applied to a model, verifies its membership of a formalism by *reduction*. This is similar to the syntax checking based on (context-free) grammars used in programming language compiler compilers. Note how

this approach can be used to model type inferencing and other more sophisticated checks.

Both types of meta-models (type-graph or grammar) can be *interpreted* (for flexibility and dynamic modification) or *compiled* (for performance).

Note that when meta-modeling is used to synthesize interactive, possibly visual modeling environments, we need to model *when* to check whether a model belongs to a language. In *free-hand* modeling, checking is only done when explicitly requested. This means that it is possible to create, during modeling, syntactically incorrect models. In *syntax-directed* modeling, syntactic constraints are enforced at all times during editing to prevent a user from creating syntactically incorrect models. Note how the latter approach, though possibly more efficient, due to its incremental nature –of construction and consequently of checking– may render certain valid models in in the modeling language unreachable through incremental construction. Typically, syntax-directed modeling environments will be able to give suggestions to modelers whenever choices with a finite number of options present themselves.

The advantages of meta-modeling are numerous. Firstly, an *explicit* model of a modeling language can serve as *documentation* and as *specification*. Such a specification can be the basis for the *analysis* of properties of models in the language. From the meta-model, a modeling environment may be *automatically* generated. The flexibility of the approach is tremendous: new languages can be designed by simply *modifying* parts of a meta-model. As this modification is explicitly applied to models, the relationship between different variants of a modeling language is apparent. Above all, with an appropriate meta-modeling tool, modifying a meta-model and subsequently generating a possibly visual modeling tool is orders of magnitude *faster* than developing such a tool by hand. The tool synthesis is *repeatable* and *less error-prone* than hand-crafting.

As a meta-model is a model in an appropriate modeling language in its own right, one should be able to meta-model that language's abstract syntax too. Such a model of a meta-modeling language is called a *meta-meta-model*. This is depicted in Figure 1.3. It is noted that the notion of "meta-" is relative. In principle, one could continue the meta- hierarchy ad infinitum. Luckily, some modeling languages can be meta-modeled by means of a model in the language itself. This is called *meta-circularity* and it allows modeling tool and language compiler builders to *bootstrap* their systems.

A model m in the Abstract Syntax set (see Figure 1.3) needs at least one concrete syntax. This implies that a concrete syntax mapping function κ is needed. κ maps an abstract syntax graph onto a concrete syntax model. Such a model could be textual (*e.g.,* an element of the set of all Strings), or visual (*e.g.,* an element of the set of all the 2D vector drawings). Note that the set of concrete models can be modeled in its own right. It is noted that grammars may be used to model a visual concrete syntax. Also, concrete syntax sets will typically be re-used for different languages. Often, multiple concrete syntaxes will be defined for a single abstract syntax, depending on the user. If exchange between modeling tools is intended, an XML-based textual syntax is often used. If in such an exchange, space and performance is an issue, an binary format may be used instead. When the formalism is graph-like as in the case of a circuit diagram, a visual concrete syntax is often used for human consumption. The concrete syntax of complex languages is however rarely entirely visual. When for example equations need to be represented, a textual concrete syntax is more appropriate.

Finally, a model m in the Abstract Syntax set (see Figure 1.3) needs a unique and precise meaning. As previously discussed, this is achieved by providing a Semantic Domain and a semantic

mapping function [[.]]. This mapping can be given informally in English, pragmatically with code or formally with model transformations. Natural languages are very ambiguous and not very useful since they cannot be executed. Code is executable, but it is often hard to understand, analyze and maintain. It can be very hard to understand, manage and derive properties from code. This is why formalisms such as Graph Grammars are often used to specify semantic mapping functions in particular and model transformations in general. Graph Grammars are a visual formalism for specifying transformations. Graph Grammars are formally defined and at a higher level than code. Complex behavior can be expressed very intuitively with a few graphical rules. Furthermore, Graph Grammar models can be analyzed and executed. As efficient execution may be an issue, Graph Grammars can often be seen as an executable specification for manual coding. As such, they can be used to automatically generate transformation unit tests.

1.3.3 Computer Automated Multi-Paradigm Modeling

Computer Automated Multi-Paradigm Modeling (CAMPaM) [VdL03] aims to simplify the modeling of complex systems by supporting

- *Multi-Formalism modeling*, concerned with the coupling of and transformation between models described in different formalisms; and
- *Model Abstraction*, concerned with the relationship between models at different levels of abstraction;

This is achieved through *Meta-Modeling* as well as the explicit modeling of *Model Transformation*.

Meta-modeling can help in defining high abstraction level notations. With meta-modeling, we can describe, using a high-level, visual notation, the (possibly graphical) syntax of languages for particular needs: domain-specific visual languages. Such languages have the potential to greatly increase system quality and reduce development costs, as they are notations tailored to specific needs.

Some languages such as the UML are rigorously defined through meta-modeling. But meta-modeling the syntax of a language is only one side of the coin. One needs to formally specify the semantics of a language as was mentioned before. We may be interested in defining a language's *operational* semantics (*i.e.,* how models described in the language are simulated or executed), or its *denotational* or *transformational* semantics (*i.e.,* defining a mapping onto another well-defined language; this may include code generation when mapping onto a virtual machine). We may also wish to *optimize* the models (*i.e.,* reduce the complexity without removing salient features). As models, meta-models and meta-metamodels can all be described as attributed, typed graphs, Graph Grammars can be used as a formal, graphical and high-level notation to specify the model transformations.

AToM^3, A Tool for Multi-formalism and Meta-Modeling, developed in the Modeling, Design and Simulation Lab (MSDL) of McGill University by Juan de Lara and Hans Vangheluwe [dLV02a], implemented these meta-modeling and graph transformation concepts. The tool was proven to be very powerful, allowing the meta-modeling of known formalisms such as DEVS [PB03], Statecharts [BV03, Fen03], UML Class Diagrams and Activity Diagrams [dLV05], Finite State Automata [VdL02], Petri Nets [dLV02b], GPSS [dLV02d], Process Interaction Networks [dLV04], Hybrid Systems [LJVdLM04, dLGV04, dLVAM03], Causal Block Diagrams [PdLV02, dLVA04b], Dataflow Diagrams [dLV02c] (a subset of Simulink) and many others. More importantly, many

new formalisms were constructed using the tool, such as the Traffic formalism [JdLMny].

The philosophy of AToM^3 is to *model everything* explicitly. That is, not only formalisms and transformations are modeled explicitly, but also composite types and the user interfaces of the generated tools. In fact, the entire AToM^3 tool was bootstrapped from a small kernel with code-generating capabilities.

1.4 A Model-Driven Approach to Scenario-Based Requirements Engineering

We propose a model-driven approach to scenario-based requirements engineering, as shown in Figure 1.4.

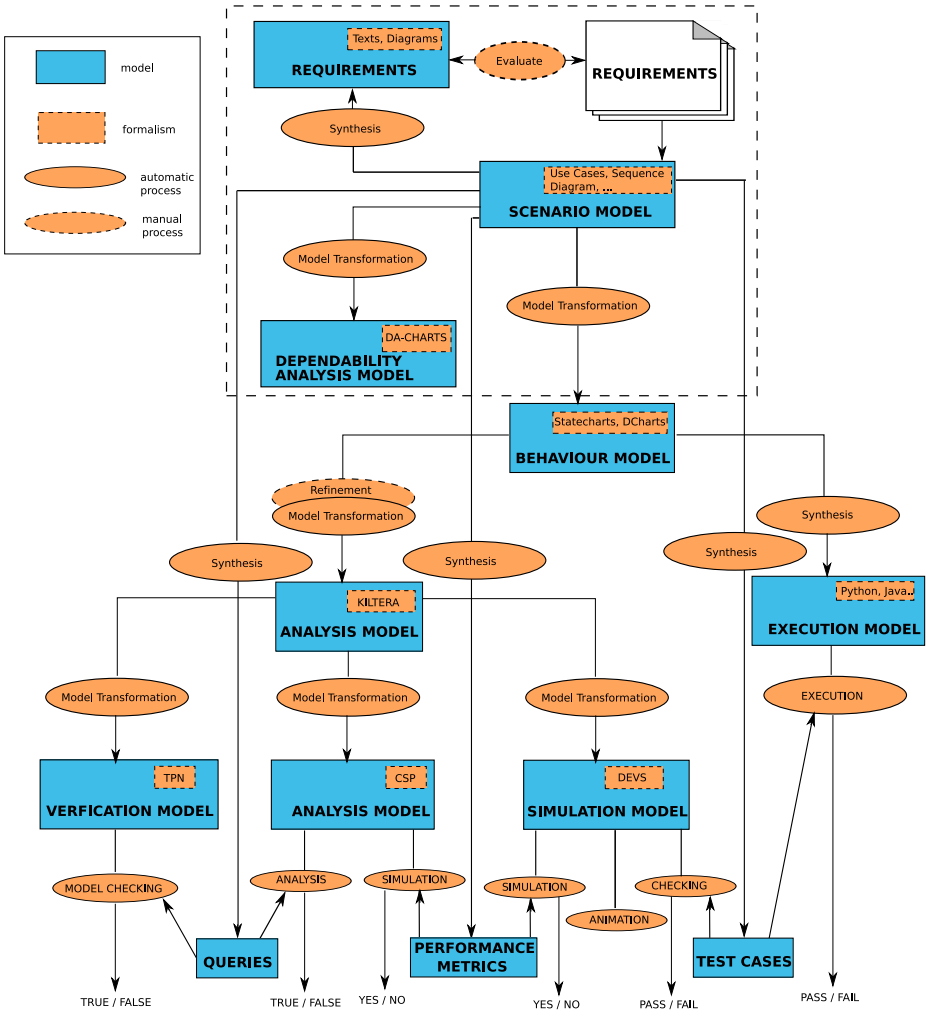

Figure 1.4: A Model-Driven Approach to Scenario-Based Requirements Engineering

The approach starts at the top level of the figure which models requirements of a system in scenarios by means of formalisms such as Use Cases, Sequence Diagrams, or Use Case Charts [Whi05]. Then, state-based behavior models (e.g., Statecharts [Har87] or DCharts [Hui04] models) can be generated automatically by model transformation. At this point, we can already use gen-

erated hierarchical state machines (HSMs) to do some automatic simulation by tools such as SCASP [Whi05], RHAPSODY [HKP05], and SVM [Fen03], and even synthesize code for execution by tools such as the Statechart compiler SCC [Hui04]. However, HSMs are limited to the cases where explicit timing information or interactions between components are important for simulation, analysis and verification. So we further transform these HSMs to models in formalisms such as DEVS, Communicating Sequential Processes (CSP), or Timed Petri Nets (TPN). Some examples can be found in [dLV02b, BV03]. We can now map HSMs into models of one single formalism, kiltera [Ern07], which allows mapping onto other formalisms as shown in Figure 1.4. Since the mapping from HSMs to kiltera is not trivial, we will still need human intervention to aid in refinement.

Kiltera aims to provide the means to model complex, dynamic, possibly large, interacting systems which have a structure that changes over time. As kiltera combines the ability to observe the passage of time and describe a system's behavior in a time-dependent manner, with the ability to describe changes in the network of communications between components, it is easy to derive models for simulation, analysis or verification from Kiltera by model transformation.

Then we can do model checking, analysis, simulation or animation with those derived models. Another advantage of our approach is that the inputs of these tasks, such as queries, performance metrics and test cases, can be automatically generated from the scenario models done at the beginning. Furthermore, as illustrated at the top of Figure 1.4, the scenario models can be used to generate textual or graphical representation of requirements in a language the customers are familiar with for their evaluation and immediate feedback.

As shown at the top of Figure 1.4, there is a transformation which leads to a dependability analysis model. This shows that it is easy to transform scenario models to some formalism such as DA-Charts for some specific analysis (e.g., dependability analysis) [MSKV06]. A detailed example will be given in a case study in Section 4.

The key to the success of our approach is the application of meta-modeling and model transformation as introduced in previous sections.

This thesis implements some key parts of the approach, in particular for those steps enclosed in the dashed rectangle on the top of Figure 1.4. Specifically, the main contribution of this thesis is the implementation of a visual modeling environment for building scenario models, for automatic transformation to behavior models for further analysis, and for automatic generation of requirements text from scenario models.

2

Sequence Diagrams Meta-Modeling in AToM^3

The following sections illustrate how to build a visual modeling environment for Sequence Diagrams with AToM^3.

First, we give a brief introduction to Sequence Diagrams formalism. Then we define the meta-model using Class Diagrams which provides the abstract syntax (denoting entities of the formalism, their attributes, relationships and constraints) as well as the concrete visual syntax (how the entities and relationships should be rendered in a visual interactive tool, as well as the possible layout constraints). Finally, we model the reactive behavior of formalism-specific user-interfaces be means of the new framework developed in Denis Dubé's M.Sc. thesis [Den06].

This new framework greatly simplifies the building of a visual modeling environment for a domain-specific formalism. Rather than purely hard-coding the environment, the reactive behavior of the visual modeling environment, including formalism-specific behaviors and layout considerations are explicitly modeled. Subsequently, the interactive environment is synthesized.

2.1 Sequence Diagrams Basics

Sequence Diagrams are a visual modeling language, which is the UML variant of Message Sequence Charts, used primarily to show the interactions between objects in the sequential order that those interactions occur. One of the primary uses of sequence diagrams is in the transition from requirements expressed as scenarios to the next and more detailed level of refinement. Scenarios are often refined into one or more sequence diagrams.

The main purpose of a Sequence Diagram is to define event sequences that result in some desired outcome. The focus is less on the messages themselves and more on the order in which messages occur. Nevertheless, most sequence diagrams will communicate what messages are sent between a systems objects as well as the order in which they occur. The diagram conveys this information along the horizontal and vertical dimensions: the vertical dimension shows, top down, the time sequence of messages/calls as they occur, and the horizontal dimension shows, left to right, the object instances that the messages are sent to/from.

- *Lifeline.* When drawing a sequence diagram, lifeline notation elements are placed across the top of the diagram. Lifelines represent either roles or object instances that participate in the sequence being modeled. Lifelines are drawn as a box with a dashed line descending from the center of the bottom edge. The lifelines name is placed inside the box.

  The UML standard for naming a lifeline follows the format:

$$InstanceName : ClassName$$

- *Messages.* The first message of a sequence diagram always starts at the top and is typically located on the left side of the diagram for readability. Subsequent messages are then added to the diagram slightly lower than the previous message.

  To show an object (i.e., lifeline) sending a message to another object, one draws a line to the receiving object with a solid arrowhead (if a synchronous call operation) or with a harpoon arrowhead (if an asynchronous signal). The message/method name is placed above the arrowed line. The message that is being sent to the receiving object represents an operation/method that the receiving objects class implements.
- *Combined Fragments.* In most sequence diagrams, the UML1.x "in-line" guard is not sufficient to handle the logic required for a sequence being modeled. This lack of functionality was a problem in UML1.x. UML2.0 has addressed this problem by removing the "in-line" guard and adding a notation element called a Combined Fragment. A combined fragment is used to group sets of messages together to show conditional flow in a sequence diagram. The UML2.0 specification identifies eleven interaction types for combined fragments. In this thesis we model four of them which are mostly used: *alternatives*, *option*, *parallel* and *loop*.

  - *Alternatives.* Alternatives are used to designate a mutually exclusive choice between two or more message sequences. Alternatives allow the modeling of the classic "if then else" logic. The word "alt" is placed inside the frames namebox. The larger rectangle is then divided into Interaction Operands. In UML, Operands are separated by a dashed line. Each operand is given a guard to test against, and this guard is placed towards the top left section of the operand on top of a lifeline. If an operand's guard equates to "true", then that operand is the operand to follow. Alternative combination fragments are not limited to simple "if then else" tests. There can be as many alternative paths as are needed. Note that according to UML2.0, if more than one alternative is true, one of them is selected nondeterministically for execution.
  - *Option.* The option combination fragment is used to model a sequence that, given a certain condition, will occur; otherwise, the sequence does not occur. An option is used to model a simple "if then" statement. The option combination fragment notation is similar to the alternation combination fragment, except that it only has one operand and there never can be an "else" guard. To draw an option combination one draws a frame. The text "opt" is placed inside the frame's namebox, and in the frame's content area the option's guard is placed towards the top left corner on top of a lifeline. Then the options sequence of messages is placed in the remainder of the frame's content area.
  - *Loop.* Occasionally one will need to model a repetitive sequence. In UML2.0, modeling a repeating sequence has been improved with the addition of the loop combination fragment. The loop combination fragment is very similar in appearance to the option combination fragment. One draws a frame, and in the frame's namebox the text "loop" is placed. Inside the frame's content area the loop's guard is placed towards the top left corner, on top of a lifeline. Then, the loop's sequence of messages is placed in the remainder of the frame's content area.
  - *Parallel.* The parallel combination fragment element needs to be used when creating a sequence diagram that shows parallel processing activities. The parallel combina-

tion fragment is drawn using a frame, and one places the text "par" in the frame's namebox. The frame's content section is broken up into horizontal operands separated by a dashed line. Each operand in the frame represents a thread of execution done in parallel.

- *Interaction Uses*. An interaction use refers to an interaction. The interaction use is a shorthand for copying the contents of the referred interaction to where the interaction use is. An interaction use is drawn using a frame. The text "ref" is placed inside the frame's namebox, and the name of the sequence diagram being referenced is placed inside the frame's content area.

  A constraint for interaction use is that the interaction use must cover all lifelines of the enclosing interaction that appear within the referred interaction.

2.2 Sequence Diagrams Meta-Model

The Sequence Diagrams formalism, shown in Figure 2.1, was modeled in the Class Diagrams formalism within AToM^3. The latter formalism is similar to UML Class Diagrams, in that it has classes with attributes, associations with multiplicities, and inheritance. The main difference lies in the fact that the AToM^3 version allows one to immediately generate a formalism-specific editor, with a generic visual modeling environment, from the Class Diagram and extra information. The rectangular boxes in the class diagram become the nodes/vertices in the generated formalism. Each of them gets a name attribute that appears on or near the visual icon in the generated formalism. The nodes and the meaning of their attributes are as follows:

- *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. The notation for an Interaction in a Sequence Diagram is a solid-outline black rectangle. The Interaction name is in a pentagon in the upper left corner of the rectangle.
- *Lifeline* represents an individual participant in the Interaction. A Lifeline is shown using a symbol that consists of a black rectangle forming its "head" followed by a vertical line that represents the lifetime of the participant. 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. These attributes are displayed inside the head rectangle in the following format:

  $$instanceName : className$$

  There is no significance to the horizontal ordering of the lifelines.
- *ActionFragment* is used to hold syntactic points (called OccurrenceSpecifications in UML) at the ends of Messages. An ActionFragment is represented as a green thin rectangle that covers the Lifeline line. In our meta-model, we restrict an ActionFragment to have at most one OccurrenceSpecification on each side. Since the order of these points along a Lifeline is significant denoting the order in which these OccurrenceSpecifications will occur, we strongly recommend to use only one OccurrenceSpecification for each ActionFragment unless there is a case where the order is insignificant. The absolute distances between the OccurrenceSpecifications on the Lifeline are, however, irrelevant for the semantics.

  The ActionFragment is actually the combination of an atomic ExecutionSpecification and an OccurrenceSpecification of UML2.0. The main reason for the combination is to

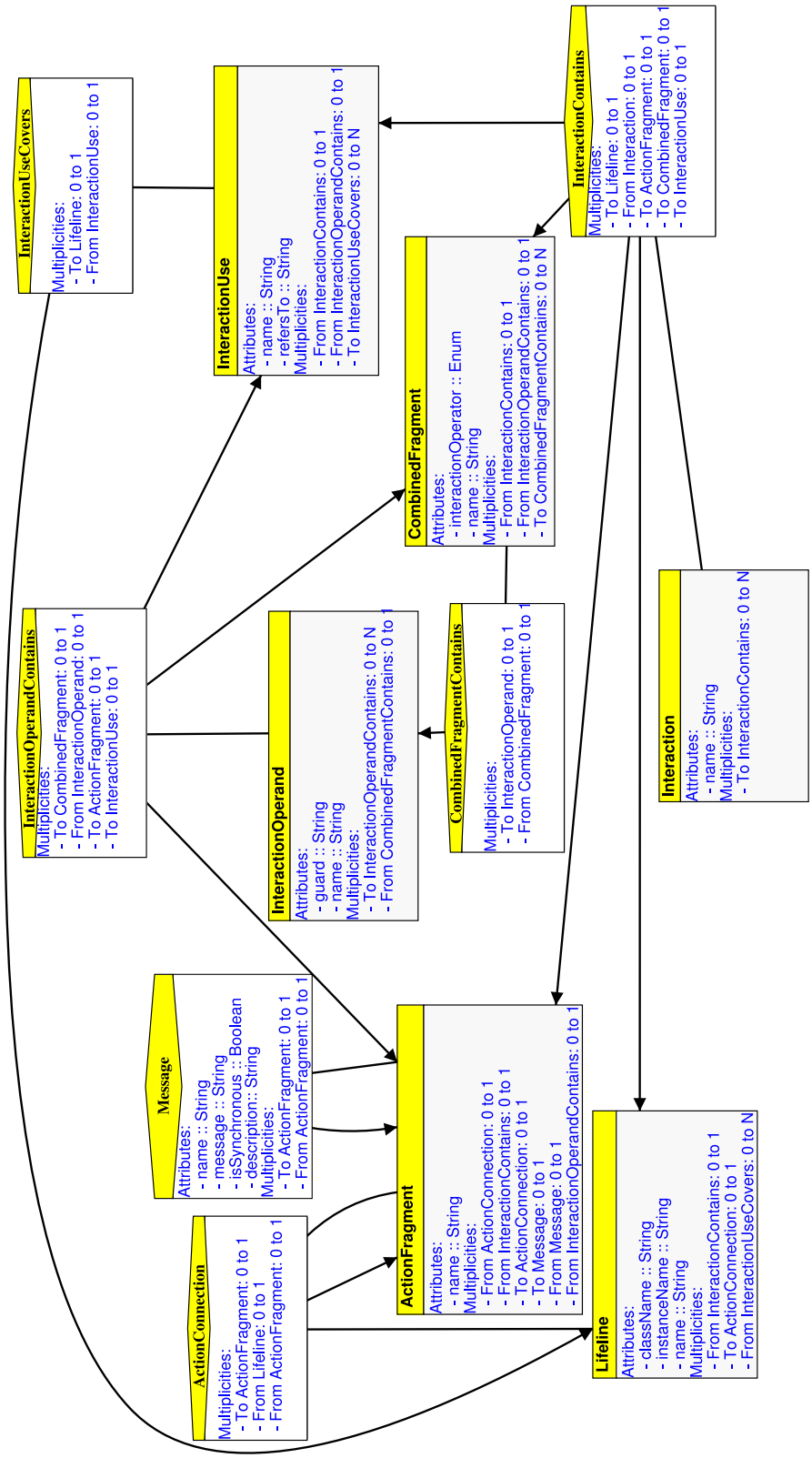

Figure 2.1: Sequence Diagrams Meta-Model in the Class Diagrams formalism

simplify the definition of the Graph Grammar rules for transforming Sequence Diagrams to Statecharts which is described in Section 3.

- *CombinedFragment* is defined by an interaction operator and corresponding interaction operands. A CombinedFragment is represented as a solid-outline black rectangle with the operator shown in the upper left corner of the rectangle. 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. An InteractionOperand is depicted as a solid-outline gray rectangle with the optional guard shown in the upper left corner of the rectangle. This visual representation is not adhere to the UML standard but makes both the graphical component layout and model transformation easier to be implemented. For example, the implementation of the UI scoping mechanism is much easier by using a rectangle than by a line.
- *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. An InteractionUse is represented as a solid-outline black rectangle with the name of the referred Interaction shown in the upper left corner of the rectangle.

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. The "description" attribute is used when one wants to give some customized description about the message. This customized description will be used when requirements text is generated from this Sequence Diagram (see details in Section 3.6).

In AToM^3 meta-modeling environment, one can specify visual representations (concrete visual syntax) for all visible entities of the generated formalism in its meta-model. Figure 2.2 lists these icons attached to each entity of the meta-model shown in Figure 2.1

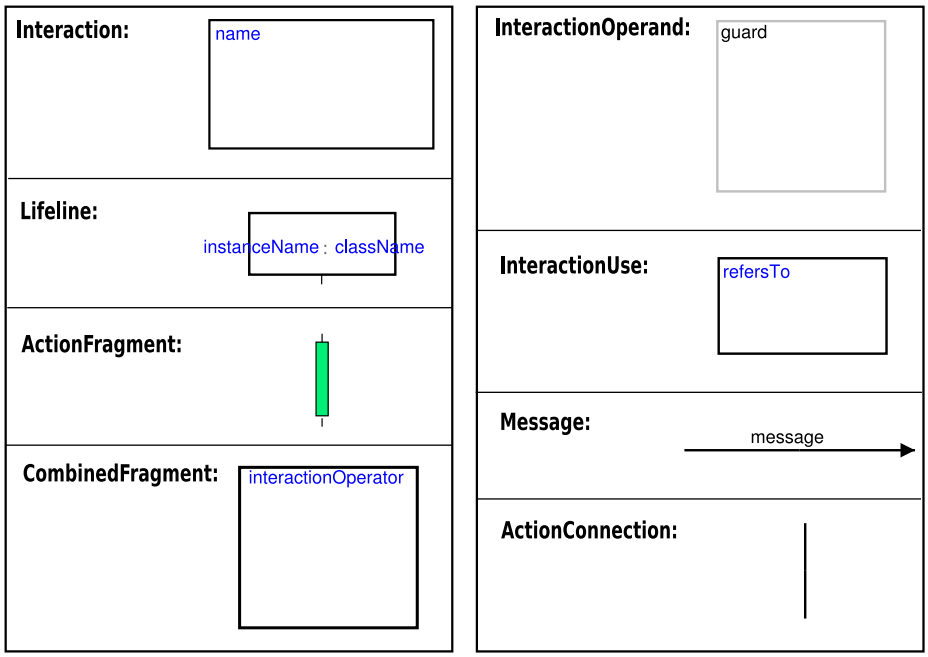

Figure 2.2: Visual representations of the meta-model of the Sequence Diagrams formalism

2.3 Formalism-Specific UI Modeling

Although the class diagram in Figure 2.1 is sufficient to generate a working modeling environment for Sequence Diagrams formalism, more formalism-specific UI modeling is needed for two main reasons.

The first reason is that the general purpose layout methods can not be directly applied to Sequence Diagrams due to the following special layout requirements. First, because the vertical dimension of a sequence diagram is the time axis. This requires proper ordering of elements along it denoting causality. Second, all entities of a sequence diagram should be aligned in both dimensions (i.e., horizontally and vertically) in aesthetic ways. Third, due to the introduction of combined fragments, a sequence diagram will have a hierarchical structure. In other words, unless a layout method is constructed for the purpose of dealing with compound graphs, graphs with hierarchical containment, the layout method will be of no use. Without automatic layout support there is a significant risk that modelers will avoid adding to an existing Sequence Diagrams model simply because the task of doing the layout manually is so time consuming.

The second reason for doing formalism-specific UI modeling is to maximally constrains users, allowing them, to only build syntactically and semantically correct models (i.e., to prevent users from constructing sequence diagrams that are illegal at the abstract syntax level). This is particularly devastating for novice modelers. For example, one could have an interaction operand directly enclosed in any other containment entities but combined fragments; or one could connect an interaction use to some lifelines which do not even exist inside the interaction that that interaction use refers to.

In general, the goal is to make the visual modeling environment as user-friendly as possible.

Thanks to Denis Dubé's thesis work [Den06], we can now use his new framework to explicitly

model the behavior of Sequence Diagrams-specific user interfaces to achieve the above goals. The framework has greatly shortened the time for building such a formalism-specific user interface.

The framework works as follows. First, a model of generic user-interface reactive behavior is constructed. The code generated from this model is generally applicable to most formalisms (i.e., it is formalism independent). Thereafter, each formalism provides additional models that refine the generic UI behaviors with more specific behaviors. Using visual cues (bounding boxes), the correct formalism-specific or generic behavior is chosen even in the presence of multiple simultaneous formalisms. Finally, the formalism-specific behaviors themselves include automatic graph layout method invocations in specific sequences and appropriate times.

We follow the approach proposed in [Den06], by altering the buttons model of the generated formalism, observing events with the use of pre and post statecharts, acting on events with Sequence Diagrams formalism and entity-specific statecharts, and handling layout for each hierarchical Sequence Diagrams entity (again with a statechart).

Furthermore, statecharts models built in [Den06] were our excellent starting point for building statecharts models for our new formalism.

Also, we follow the conventions used in [Den06] to describe these statecharts. That is, in the following subsections, the labels on the states and transitions of the UI behavior statecharts use a custom notation to make them more expressive. A star, x*, indicates that action code is present. A plus, x+, indicates that a different statechart handles the action. Parenthesis, <x>, indicate that the trigger event is generated by another statechart, such as the pre/post UI observers or another UI behavior statechart. Regular brackets, (x), indicate the event was generated by the initialization routine for the entity when it is first instantiated. Square brackets [x] indicate that the event was generated by the statechart itself, usually within the action code of a state.

2.3.1 Buttons Model

The buttons model is a trivial model, in the aptly named Buttons formalism. The buttons in this model correspond directly to the buttons that appear in the AToM^3 application's formalism toolbar. Buttons models are automatically generated from a meta-model, such as a class diagram, to allow a user to create the entities specified in the meta-model.

A more exotic change to the buttons model is also needed. The Interaction entity creation button is modified to instantiate 5 different statecharts. A statechart for controlling buttons behavior, a pre and a post statechart, an Interaction specific behavior statechart, and finally an Interaction specific layout statechart. These will be discussed further in the following subsections. The number of different statecharts may seem excessive, but Interaction is not an ordinary entity. Its purpose is to provide a formalism-specific override to the generic UI behavior.

The other entity-creating buttons, such as for creating a Lifeline, are also modified. Instead of the buttons creating the entity in question on the canvas, they directly send an event to the Interaction's button statechart. Thus, the Interaction is made fully responsible for the creation of all other entities. Due to this approach, it is impossible to create a new entity outside of the visual container formed by the Interaction.

2.3.2 Button Behavior Model

The button behavior model is quite simple and is shown in Figure 2.3. When the button to create entity X is pushed, the events "<Reset>" and "<X Button>" are sent to this statechart.

If not already there, the statechart moves to an Idle state upon receipt of the first event. The second event then moves it to a state whereby entity X can get instantiated. It then waits for an event requesting the creation of that entity. The "<Create>" event is generated by the Interaction specific behavior statechart when it intercepts and handles the "Model Action" event. See Section 2.3.4 for more details.

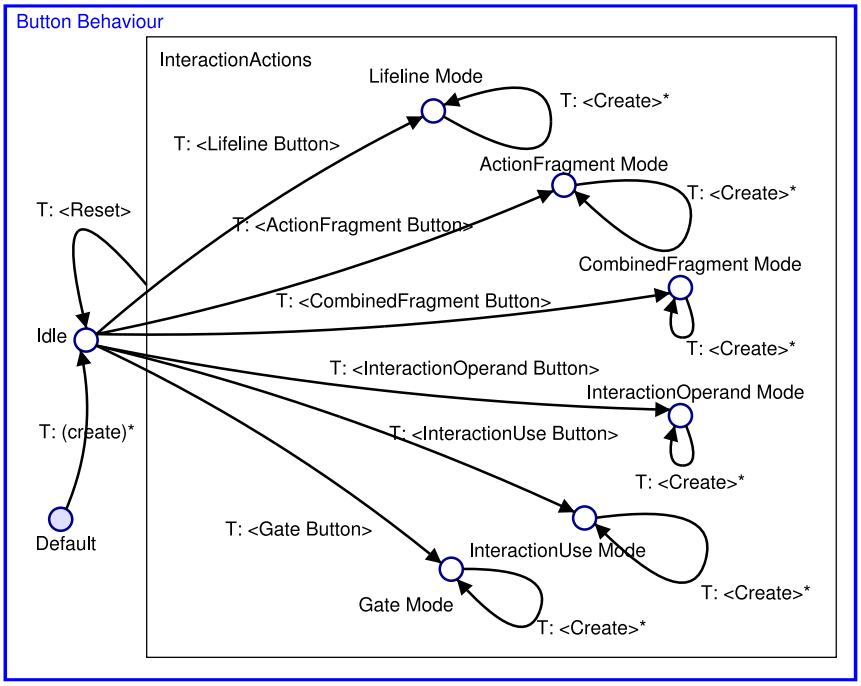

Figure 2.3: Button Behavior Statechart

2.3.3 Pre/Post Observer Statechart Models

A pre UI statechart observes events before the generic UI behavior statechart acts on them. The pre UI statechart is shown in Figure 2.4. Likewise, the post UI statechart observes events just after the generic UI behavior statechart acts on them and is shown in Figure 2.5. For the Sequence Diagrams formalism, these observers prove useful mainly for the following four events: deletion, selection, the drop after dragging selected entities, edit, and save.

The deletion event is useful for two reasons. The first of these is rather trivial. It removes the pre/post statecharts from the main event loop if the Interaction instance has been deleted. The second is a layout consideration. If an action fragment is deleted, then its parent, an interaction operand or the interaction, may require less area. Thus it makes sense to send the behavior statechart of the parent a layout request event. The parent's behavior chart will in turn send a layout request event to the parent's layout statechart, where the layout will finally be handled. At the very least, this layout will result in the parent container shrinking itself to occupy less space. Ultimately, the parent may also completely redraw itself and its contents in a new configuration that takes advantage of the state's removal.

The selection event makes it possible to detect what entities are selected. If an entity with hierarchical children, via containment relationships, is selected, then the children should also be selected. This makes it impossible for a user to delete or drag a container entity without

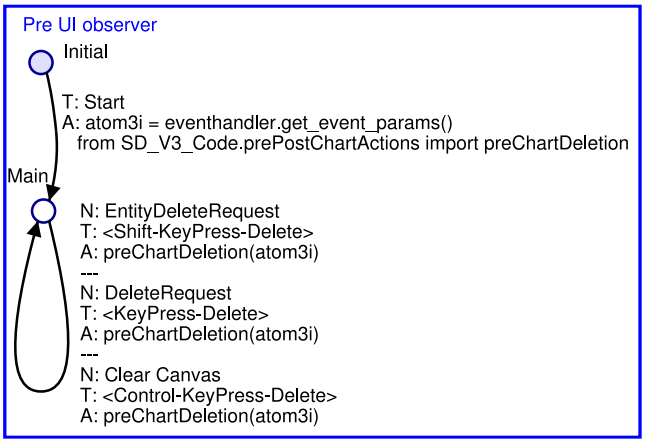

Figure 2.4: Pre UI Observer Statechart

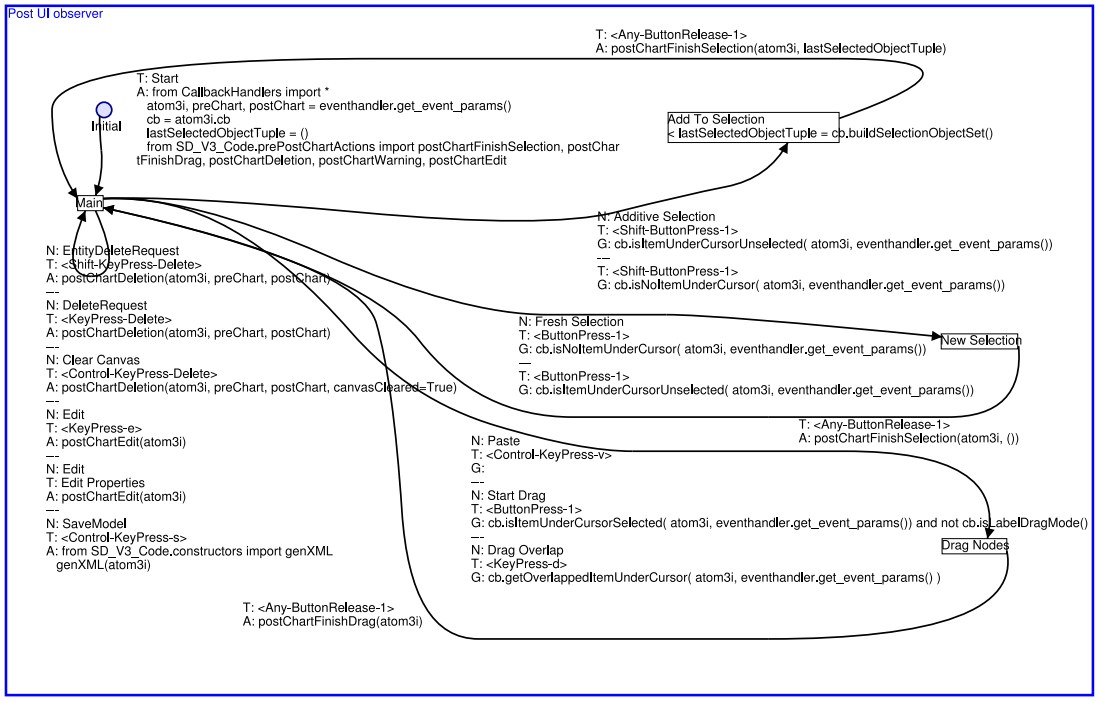

Figure 2.5: Post UI Observer Statechart

doing the same to the contained children entities.

The drop event at the end of a drag and drop operation allows for dragging multiple entities outside the Interaction scoped UI environment. This is very useful for temporary editing by a modeler. For example, one could drag a combined fragment out of the Interaction, which triggers a containment disconnect (as the next section will show), and then drag the combined fragment back into another interaction operand inside the Interaction, triggering a new containment relation inside an interaction operand.

The save event triggers a procedure to generate an XML file which stores some basic information about the current Interaction model (e.g., in which file the model is stored, what lifelines it contains, and etc.). This information is used for validation when the model is referred to by an interaction use in another model. For example, an interaction use is valid if the lifelines it covers exist in the referred interaction model (see Section 2.3.4).

Finally, the edit event simply triggers an edit dialog. It could have been handled with entity-specific behavior statecharts. However, if edit were only handled by the Interaction behavior statechart, then an entity outside the containment of Interaction could not be edited.

2.3.4 Formalism Entity-Specific Behavior Models

All visual entities of the Sequence Diagrams formalism require their own behavior models. The most important of these are those associated with the artificial entity that contains all others of the Sequence Diagrams formalism and those of the combined fragments and interaction operands. Referring to the class diagram in Figure 2.1, these are Interaction, CombinedFragment and InteractionOperand respectively. At the other extreme, the behavior statechart for Message, is trivial. All the remaining entities, excluding the non-visual containment relationships, use behavior statecharts that are subsets of that of the interaction operand.

Interaction Behavior Statechart

The behavior of the Interaction (see Figure 2.6) entity begins with initialization when the entity is first created. This initialization includes a "(create)*" trigger that sets the active state to "Idle". From then on, the following four events trigger interesting behavior:

1. The "<InteractionSelect>*" event is generated by the post UI observer statechart. The event indicates that Interaction has been selected by the user. It is then necessary to ensure that all the hierarchical children of Interaction are also selected so that delete and drag operations work as expected.
2. The "<Control-Button-Press-3>" event is directly captured from the main event loop and explicitly handled, thus halting its propagation. This event indicates that a Sequence Diagrams formalism entity should be added to the canvas. Note that the same event is generated if one uses the AToM^3 menu system or a keyboard/mouse shortcut. The actual creation of an entity is of course handled by the button behavior statechart previously seen in Section 2.3.2.
3. The "<Control-Button-Press-1>*" event is also directly captured from the main event loop. Moreover, this event triggers a lock, forcing all events in the main event loop to only this statechart. The lock is only released when either an arrow is finally created or the process is aborted, via the "<Arrow Created>*" and "Reset*" events respectively. It is necessary to refine the behavior found in the generic UI behavior statechart for two reasons. The first is merely for the convenience of the user. Instead of allowing the user to

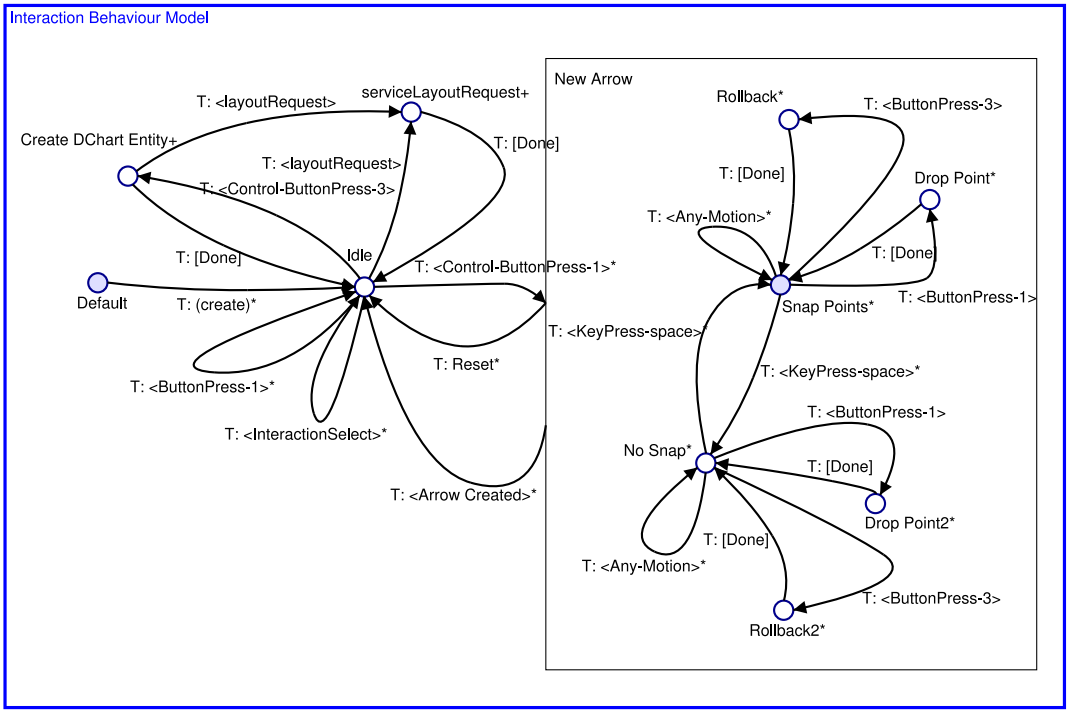

Figure 2.6: Interaction Behavior Statechart

draw arrows to indicate containment relationships, only transitions may be drawn. This saves time, and a perfectly good drag-and-drop method exists for creating and destroying containment relationships as shall be shown later in this section. The second reason is simply to know when transitions are actually created so that their UI behavior statecharts may be initialized.

The "<Arrow Created>*" event also invokes a procedure which aligns a lifeline and all action fragments along it, if the created arrow is an ActionConnection (a link between a lifeline and an action fragment or between two action fragments), or aligns an interaction use and a lifeline it covers if the created arrow is of type InteractionUseCovers.

4. The "<layoutRequest>" event is generated exclusively by the UI behavior statecharts of the children entities of Interaction. This event occurs when a new entity is created since the new entity will be contained by the Interaction and thus upsets the old layout. The event can also occur when Interaction is idle, such as when an entity is manually dragged by the user. The layout request is forwarded to the layout statechart of the Interaction, described in Section 2.3.4.

   Before the layout request is forwarded, two formalism-specific layout procedures are done. First, each lifeline and all action fragments along it are aligned vertically. Second, all visual links (i.e., links of types of ActionConnection and Message) are straightened out.

CombinedFragment Behavior Statechart

The behavior of the CombinedFragment (see Figure 2.7), is the most complex of all. Fortunately, it is also re-usable by many other entities as shall be shown further on in this section.

The initialization phase is rather involved, with two main possibilities. The first is that an interactive session with the user is in effect, in which case the "(create)" trigger signals the creation of a new Interaction. Immediately, the user is presented with a dialog asking him in which of the entities in the region of the newly created CombinedFragment they would like to contain the new fragment. If the fragment is successfully connected to either an Interaction or another InteractionOperand, then the "[didConnect]" trigger is generated, followed by a "<layoutRequest>" event to the container, and finally a "[Done]" event to set the state to "HasParent". If the fragment is not successfully connected, then a "[didNotConnect]" event is generated and the active state is set to "NoParent".

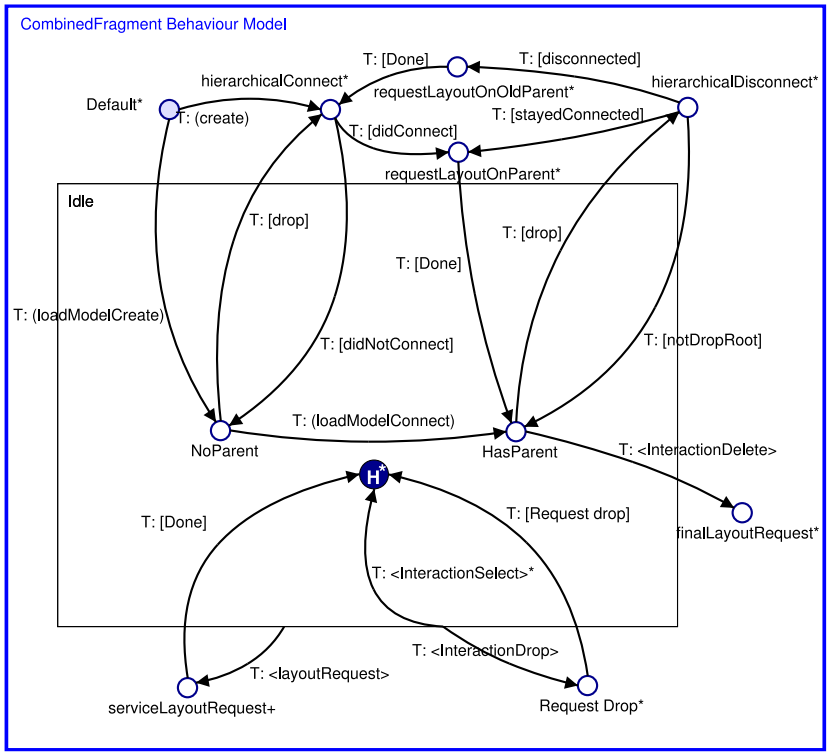

Figure 2.7: CombinedFragment Behavior Statechart

Finally, the second of the two possibilities is that the model was being loaded rather than interactively edited. In this case, a "(loadModelCreate)" event is first sent when the CombinedFragment is first instantiated, setting the active state to "NoParent". Then a second "(loadModelCreate)" event is sent if a containing relationship is instantiated with this CombinedFragment as its parent, thus setting the active state to "HasParent". The following is a list of all the events that occur after the initialization phase. Unless stated otherwise, the events are generated by the post UI observer statechart.

1. The "<InteractionSelect>*" event is dealt with in the same manner as the Interaction UI behavior statechart. All hierarchical children are selected.
2. The "<InteractionDrop>" event indicates that this fragment, among potentially many other entities, has just been dragged and then dropped. The transition with this trigger promptly generates two events: "[Done]", which restores the active state to either "No-

Parent" or "HasParent", followed by "[drop]", which causes hierarchical connection or hierarchical disconnection, respectively, to be attempted. A disconnection occurs only if the entity has been dropped outside of its parent container and the user has explicitly agreed to disconnect it. This triggers a "<layoutRequest>" followed by an attempt to hierarchically connect the disconnected fragment in its new location.

3. The "<InteractionDelete>" event indicates that this fragment is to be deleted. Before being erased, it warns its hierarchical container parent with a "<layoutRequest>". In this fashion the parent can find a new layout that takes advantage of the extra space afforded by the deleted entity.
4. The "<layoutRequest>" event is generated exclusively by the UI behavior statecharts of the children entities of CombinedFragment, just as it was in Interaction. This event occurs whenever the children of this entity are modified by the user, such as by add/removing them from the CombinedFragment or by simply moving them. The layout request is forwarded to the layout statechart of the CombinedFragment, described in Section 2.3.4.

Message Behavior Statechart

The behavior of the Message is trivially simple, as Figure 2.8 shows. As noted earlier, the message is a hyper-edge only in the meta-model, in the generated Sequence Diagrams formalism itself it is a simple directed edge with one source and one target. The transition is first initialized with a "(create)" event. Afterwards, it simply awaits "<Edit>*" events from the post UI behavior chart in order to apply changes made in its edit dialog. These changes affect the information content of the label associated with the message.

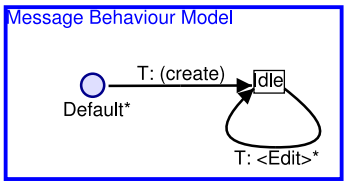

Figure 2.8: Message Behavior Statechart

Other Behavior Statecharts

The UI behavior statecharts of the remaining Sequence Diagrams formalism entities are subsets of the one previously shown for CombinedFragment. The behavior statechart of InteractionOperand entity which is also a hierarchical container entity, has the same structure of that of CombinedFragment.

The remaining entities are not hierarchical container entities, but rather primitive children. They include Lifeline, ActionFragment, and InteractionUse. The main structural difference between their UI behavior statecharts and that of the one for CombinedFragment is that they do not accept the "<layoutRequest>" event. Naturally, no entity exists that would generate and send it to them.

Another difference the InteractionUse UI behavior statechart has is it accepts "<Edit>" event. The event indicates that the user has opened an edit dialog on the InteractionUse attributes. In particular, the user may give or change the name of the referred interaction. Thus the transition with this trigger event will execute action code to load the XML file of the referred

interaction. The XML file stores basic information about the Interaction model it represents (e.g., in which file the model is stored, what lifelines it contains, and etc.) and is generated when the model is saved (see Section 2.3.3). Then, the information retrieved from the XML file is used for validation. First, an interaction use is valid if the lifelines it covers exist in the referred interaction model. Second, an interaction can not contain an interaction use which refers to the interaction itself to avoid infinite looping.

In all other aspects, the UI behavior statecharts are identical to that of the CombinedFragment.

Layout Behavior

We reuse one of the layout behavior statechart models in [Den06], the one using the built-in Force-transfer layout algorithm, as the basic layout behavior model of the Sequence Diagrams formalism. The model is shown in Figure 2.9. Structurally, it is very simple. It is initialized with a "(create)*" event and is thereafter ready to do layout. The "<applyLayout>" event, which triggers the layout sequence, is generated by the behavior statechart associated with the entity requiring layout. This occurs when the entity's behavior statechart enters the state "serviceLayoutRequest+". Thereafter, the following sequence of actions occur:

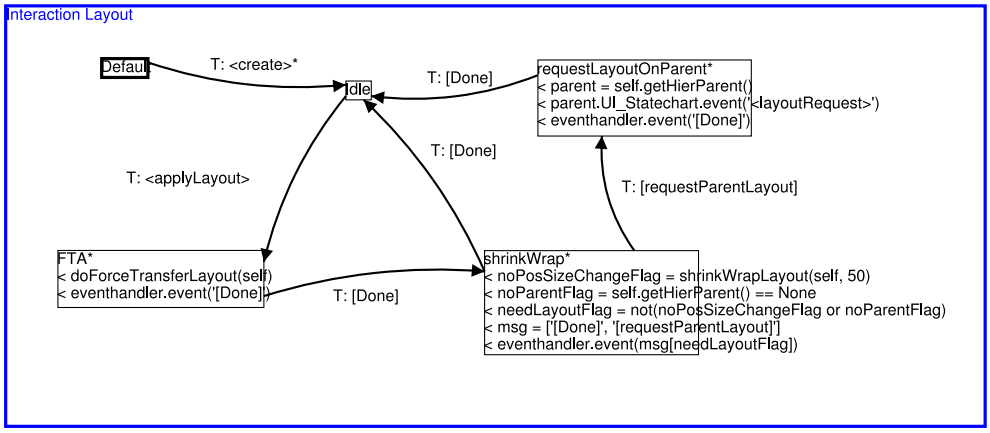

Figure 2.9: Force-Transfer Layout Statechart

1. Apply the built-in general purpose layout algorithm. Inside the action code, a choice is made of which types of entities and links that should be sent to the layout algorithm. This choice is generally limited to only the direct children entities of the hierarchical container parent and the visual arrows between them. Moreover, all the parameters passed to the layout algorithm are chosen. After the layout is applied, a "[Done]" event is generated.
2. Apply a trivial shrink-wrapping algorithm. This simply fits the hierarchical parent to be visually just large enough to contain all its children entities. As is explained in the third action, either a "[Done]" or a "[requestParentLayout]" event are generated when done.
3. Send a layout request to the behavior statechart of the parent of this hierarchical container. This action is only taken conditionally, which is depicted in the layout statechart using two alternate transitions. Clearly one condition is that the hierarchical container possesses a hierarchical parent itself. The other condition is that the hierarchical container has either moved or changed size. Obviously, if neither position nor size have

changed, the layout of the higher-levels of the hierarchy are completely unaffected. Finally, a "[Done]" event is generated and the layout statechart returns to the "Idle", ready state.

The propagation of layout requests described in action 3 propagate upwards only, in the current implementation. In other words, they propagate from the lowest to the highest level of the hierarchy. This is because the lowest level of the hierarchy will determine a layout that uses a certain area. Requesting this lower level hierarchical container to use more space is meaningless, since area is at a premium. Requesting it to use less space is equally unfeasible, since it will simply result in overlap. By obscuring information, overlap defeats the purpose of automatic layout.

2.4 Sequence Diagrams Modeling in AToM^3

Figure 2.10 shows the visual environment of the Sequence Diagrams formalism built in AToM^3. The buttons on the top panel give access to all the entities to be used in a Sequence Diagram model.

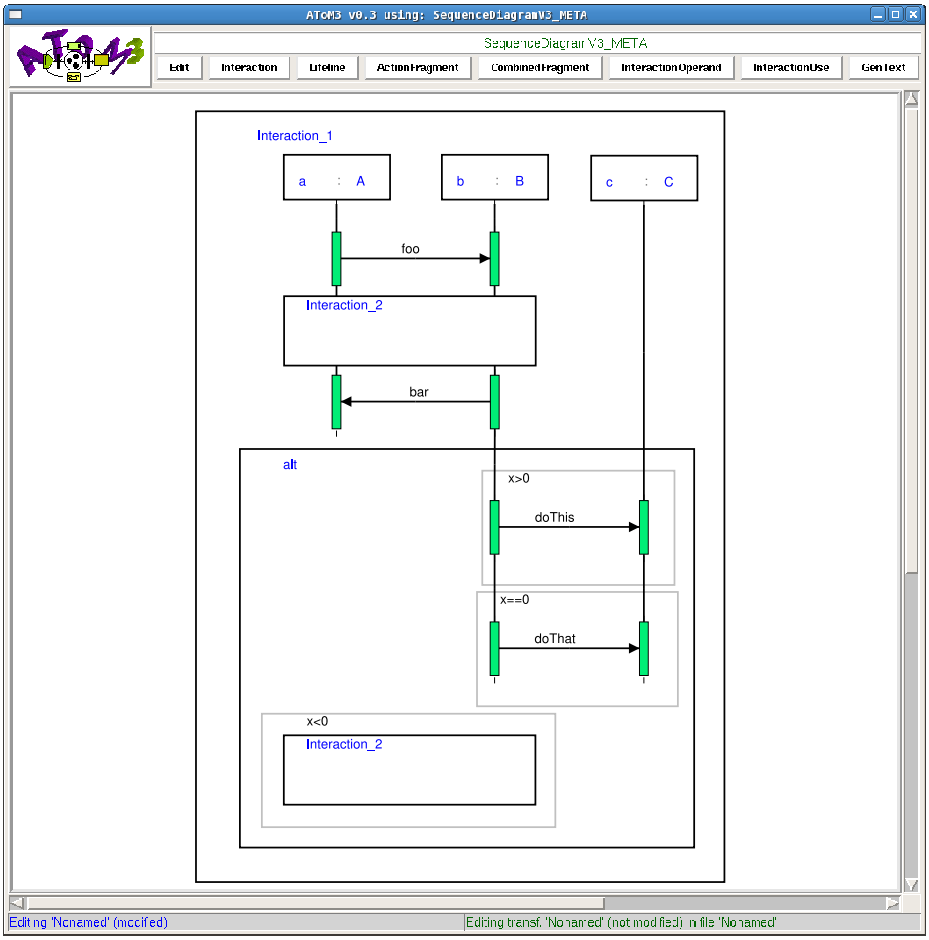

Figure 2.10: Sequence Diagrams modeling environment built in AToM^3

The illustrated Sequence Diagram model contains one interaction (and only one, as constrained

by the meta-model) which encloses three lifelines: $a : A$, $b : B$, $c : C$. There is an interaction use between message foo and bar which refers to interaction $Interaction\_2$ (containing lifelines $a : A$ and $b : B$ and a series of messages between $a : A$ and $b : B$ (not shown in this model)). An "alternative" combined fragment follows message bar which contains three operands: $x > 0$, $x == 0$ and $x < 0$. Each of the first two operands, $x > 0$ and $x == 0$, contains one message sending from $b : B$ to $c : C$. Inside the last operand, in case of $x < 0$, the same behavior which is modeled in $Interaction\_2$ and represented by the second interaction use repeats.

3
Model Transformation

3.1 Introduction

The transformation of models is a crucial element in all model-based endeavors [VdL04]. 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 Grammar formalism. Graph grammars are a natural, formal, visual, declarative and high-level representation of the transformation. A Graph Grammar 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 are found.

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. This is cleaner than adding "helper" relationships to either of those two formalisms or than using some of the relationships of those two formalisms out-of-context.

3.2 Sequence Diagrams Model to Statecharts Model

The Sequence Diagrams formalism is used primarily to show the interactions between objects by means of messages arranged in time sequence. It has become one of the most important tools in scenario-based requirements engineering process, as it can be used to capture and elicit dynamic and functional behaviors of a system, in a very readable but also quite formal way. One of the primary uses of sequence diagrams is, during the requirements phase of a project, to transition from requirements expressed as use cases to the next and more detailed level of refinement. Then during the design phase, architects and developers can use the diagram to force out the system's object interactions and the overall system design. Not only analysts, architects and developers can benefit from the diagram, but a business staff can also find sequence diagrams useful to communicate.

Statecharts are the essence of executable models. 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 [DH01] and Use Case Charts [Whi05]) to Finite State Machines (e.g., Statecharts) has become one of the key activities in object-oriented analysis and design. The generated state machines can be simulated by tools such as Rhapsody [Rha] and SVM [Hui04].

Since the release of the new UML specification, UML 2.0 [MLS05], the Sequence Diagrams formalism has become more powerful and rigorous as it provides more well-defined constructs. For example, structured control constructs (Combined Fragments) such as loops, conditionals, and parallel execution, are introduced, which provide a more precise way to model more complex flow of control. The introduction of Interaction Uses (an interaction use is a reference to another interaction, which is usually defined in its own sequence diagram) provides a natural way to reuse 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, e.g., 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 (using existing graph transformation engines). 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 sections will explain the strategy and algorithms for tackling these problems in details.

3.2.1 Four Phases of the 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. Figure 3.1 depicts these four phases, which will be explained in the subsequent sections. Now they are summarized here.

1. *Initialization.* In this phase, the source model is parsed, and some global and local data

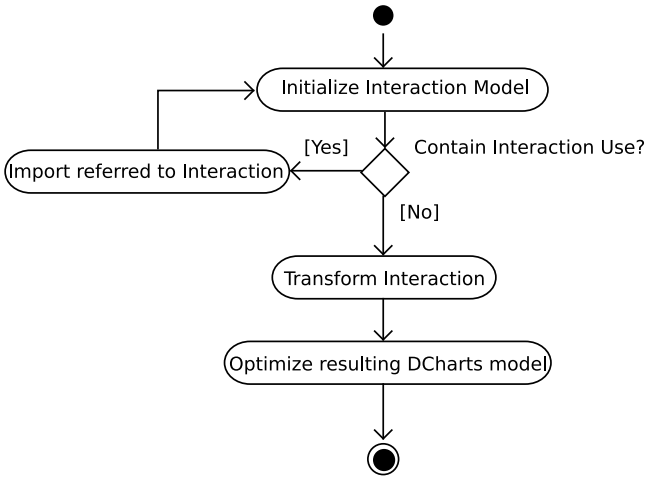

Figure 3.1: Four phases of the transformation

structures which are 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 meta-model of the Sequence Diagrams formalism) is checked when the model is saved (see Section 2.3.4). Because the importation will change the original model by adding new elements, *Initialization* is needed again after this phase.
3. *Transformation*. This is the core phase of the whole transformation. The transformation proceeds in two dimensions. The horizontal dimension follows the order of appearances of different objects. The vertical dimension follows the order of occurrences of a series of messages. The procedures ensure the processing is in the correct order with consideration of nested control constructs.
4. *Optimization*. After the previous two phases, an executable Statecharts model is generated. However, a number of redundant elements are also added to the target model in those phases, which makes it hard to read or refine. This phase is analogous to the optimization phase after the code generation of a compilation.

Figure 3.2 shows an example of Sequence Diagrams in AToM^3. We use this example to illustrate the key concepts and algorithms of the transformation process. On the left 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 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.

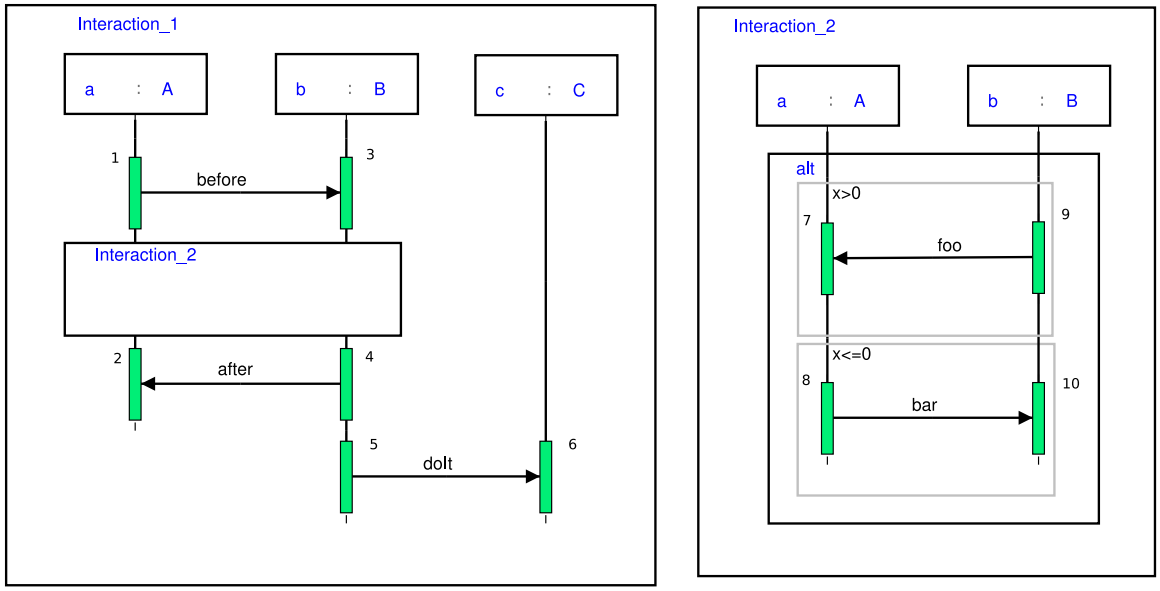

Figure 3.2: A simple example of Sequence Diagrams model in AToM^3

3.2.2 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. In general, there are three reasons why we need these data structures for the transformation procedure. First, it is because of the complex nature of the Sequence Diagrams formalism, as we discussed previously. Second, as the use of the graph-matching and the simple ordering of graph grammar rules (in AToM^3, which only supports any minimal "programmed graph rewriting") is inadequate to control the complex transformation flow in our case, if they work only by themselves. Last, it is because we want to reduce the number of the graph grammar rules which is already over forty. The evaluation and manipulation of these data structures are implemented by intensively using *pre-condition* and *post-action* of graph grammar rules.

The initialized data structures (or variables) are listed in Table 3.1 and Table 3.2. 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.

As we will explain in Section 3.2.4, the transformation of a sequence diagram proceeds in order of objects (or lifelines) from left to right, and from top to bottom along each lifeline. Two important types of information need to be known to ensure the correct ordering. First, which lifeline is currently being transformed, which is recorded by $g\_currentLifeline$. Second, the vertical ordering of messages and fragments along each lifeline (recorded in $coveredLifelinesMap$ and *position*) and what the vertical position of the current transformation is (recorded by $g\_currentPosition$).

Table 3.1: Global data structures initialized in Phase One

Name	Type	Description
g_initialized	boolean	If initialization of an Interaction is done
g_maxScope	int	Deepest level of nesting an Interaction has
g_topLevelCombinedFragments	list	List of all Combined Fragments at top level
g_transitionCount	int	Counter for the number of transitions generated so far, used to generate a globally unique name for a transition
g_stateCount	int	Counter for the number of states generated so far, used to generate a globally unique name for a state
g_currentLifeline	object	Lifeline element being transformed
g_currentPosition	int	Vertical position of a Lifeline the transformation is working on
g_currentScope	int	Nesting scope the transformation is working on
g_currentOptimizationScope	int	Nesting scope the transformation is working on during optimization
g_currentInteractionUse	object	Interaction Use being transformed
g_currentImportedInteraction	object	Interaction being imported because of an Interaction Use
g_originalInteraction	object	Interaction used to distinguish from imported ones

The $coveredLifelinesMap$ of a Combined Fragment in Table 3.2 is needed. The reason is that each Combined Fragment may cover multiple Lifelines. And for each of these Lifelines, the vertical position of the Combined Fragment is different, and a flag to remember if an iteration of the transforming process is done on that Combined Fragment is needed. The reason for the use of $coveredLifelinesMap$ of an Interaction Operand is similar, but without the need to remember the vertical position.

In order to clearly explain all graph grammar rules for the transformation of a Sequence Diagram model to a Statecharts model, we adopt the following method: first, we give a list of the rules with their order and short descriptions in a table; then we show their concrete visual syntax; and finally we describe the necessary algorithms.

Table 3.2: Local data structures initialized in Phase One

Name	Linked Element	Type	Description
coveredLifelinesMap	CombinedFragment	dict	Map of a Combined fragment to remember its vertical position on a Lifeline and whether it is processed for each covered Lifeline. Mapping: $lifeline \rightarrow [position, isProcessed]$
scope	CombinedFragment	int	Nesting scope of a Combined fragment inside an Interaction
ggParent	CombinedFragment	object	Parent element of a Combined fragment
coveredLifelinesMap	InteractionOperand	dict	Map of an Interaction Operand to remember if it's processed for each of covered Lifeline. Mapping: $lifeline \rightarrow [isProcessed]$
belongsTo	ActionFragment	object	Lifeline to which an Action fragment belongs
position	ActionFragment	int	Vertical position of an Action fragment on its root Lifeline
scope	ActionFragment	int	Nesting scope of an Action fragment inside an Interaction
scopeHasSet	ActionFragment	boolean	If the nesting scope of an Action fragment is set

There is one graph grammar rule in this phase which should be executed after the *InitialAction* of the whole graph grammar and may be executed each time after the importation of referred interactions is done ,if necessary. As there is no concrete visual syntax for this rule, we only list it in Table 3.3.

Table 3.3: Graph Grammar rules in execution order in Phase One

Order	Rule Name	Description
0	initInteraction	Initialize data structures used in the next phases to help the control of the transformation flow.

The pre-condition is very simple, just check if the global *g_initialized* is true. The main procedure of the initialization in the post-action has three steps:

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

Algorithm 1 setPosition(Graph graph)

```
for all lifeline in graph do
  count = 0
  while hasNextActionFragment(lifeline) do
    actionFragment = getNextActionFragment(lifeline)
    actionFragment.belongsTo = lifeline
    actionFragment.position = count
    if insideCombinedFragment(actionFragment) then
      combinedFragment = getParent(actionFragment)
      setCFPosition(combinedFragment, lifeline, count)
    end if
    count += 1
  end while
end for
## set position of CombinedFragment recursively
setCFPosition(CombinedFragment combinedFragment, Lifeline lifeline, int position):
if not hasSetForCurrentLifeline(combinedFragment, lifeline) then
  combinedFragment.coveredLifelinesMap[lifeline] = [position, False]
  if insideCombinedFragment(combinedFragment) then
    parentCombinedFragment = getParent(combinedFragment)
    setCFPosition(parentCombinedFragment, lifeline, position)
  end if
end if
```

We do not provide details about some procedures used in the Algorithm 1 and Algorithm 2, because their names are self-explanatory(for example, $hasNextActionFragment$, $getNextActionFragment$, etc.).

Algorithm 2 setScope(Graph graph)

```
g_maxScope = 0
interaction = getInteraction(graph)
for all combinedFragment in interaction do
  g_topLevelCombinedFragments.append(combinedFragment)
  setCFandAFScope(combinedFragment, 0)
end for
## set scope of CombinedFragment and ActionFragment recursively
setCFandAFScope(CombinedFragment combinedFragment, int scope):
if scope > g_maxScope then
  g_maxScope = scope
end if
for all containedEntity in combinedFragment do
  if isinstance(containedEntity, ActionFragment) and not containedEntity.scopeHasSet
  then
    containedEntity.scope = scope + 1
    containedEntity.scopeHasSet = True
  end if
  if isinstance(containedEntity, CombinedFragment) then
    setCFandAFScope(combinedFragment, scope+1)
  end if
end for
```

3.2.3 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 shown in Table 3.4. The procedure of the task is used in the post-action of rule *importInteraction* and the algorithm is shown in Algorithm 3.

If we apply the algorithm to the example shown in Figure 3.2, the values of the variables attached to the lifelines graph elements of $Interaction\_2$ will be set as shown in Table 3.5. Note that af stands for ActionFragment and the subscript numbers correspond to those shown in Figure 3.2.

There are thirteen rules in this phase which are shown in Figures 3.3, 3.4 and 3.5.

The list of rules, their execution order and short descriptions are given in Table 3.6. Note that some rules have the same order which can have two meanings. First, it means that there

Table 3.4: Local data structures linked with a Lifeline element in Phase Two

Name	Type	Description
headAF	object	Action fragment in the head position
tailAF	object	Action fragment in the tail position
beforeAF	object	Action fragment in the position before the interaction use referring to the enclosing interaction
afterAF	object	Action fragment in the position after the interaction use referring to the enclosing interaction

Algorithm 3 setInteractionUsePosition(Graph graph)

```
importedInteraction = getImportedInteraction(graph)
g_currentImportedInteraction = importedInteraction
for all lifeline in importedInteraction do
  lifeline.tailAF = None
  lifeline.headAF = getNextActionFragment(lifeline)
  while hasNextActionFragment(lifeline) do
    lifeline.tailAF = getNextActionFragment(lifeline)
  end while
  lifeline.beforeAF = None
  lifeline.afterAF = None
  originalLifeline = getOriginalLifeline(g_originalInteraction)
  setInteractionUsePositionHelper(originalLifeline, lifeline)
end for
## set replacement positions based-on geometrical coordinates
setInteractionUsePositionHelper(Lifeline originalLifeline, Lifeline lifeline):
(x_iu, y_iu, m_iu, n_iu) = getGraphBoundaryBox(g_currentInteractionUse)
while hasNextActionFragment(originalLifeline) do
  actionFragment = getNextActionFragment(lifeline)
  (x_af, y_af, m_af, n_af) = getGraphBoundaryBox(actionFragment)
  if n_af ≤ y_iu then
    lifeline.beforeAF = actionFragment
  end if
  if (n_iu ≤ y_af) and (lifeline.afterAF ≠ None) then
    lifeline.afterAF = actionFragment
  end if
end while
```

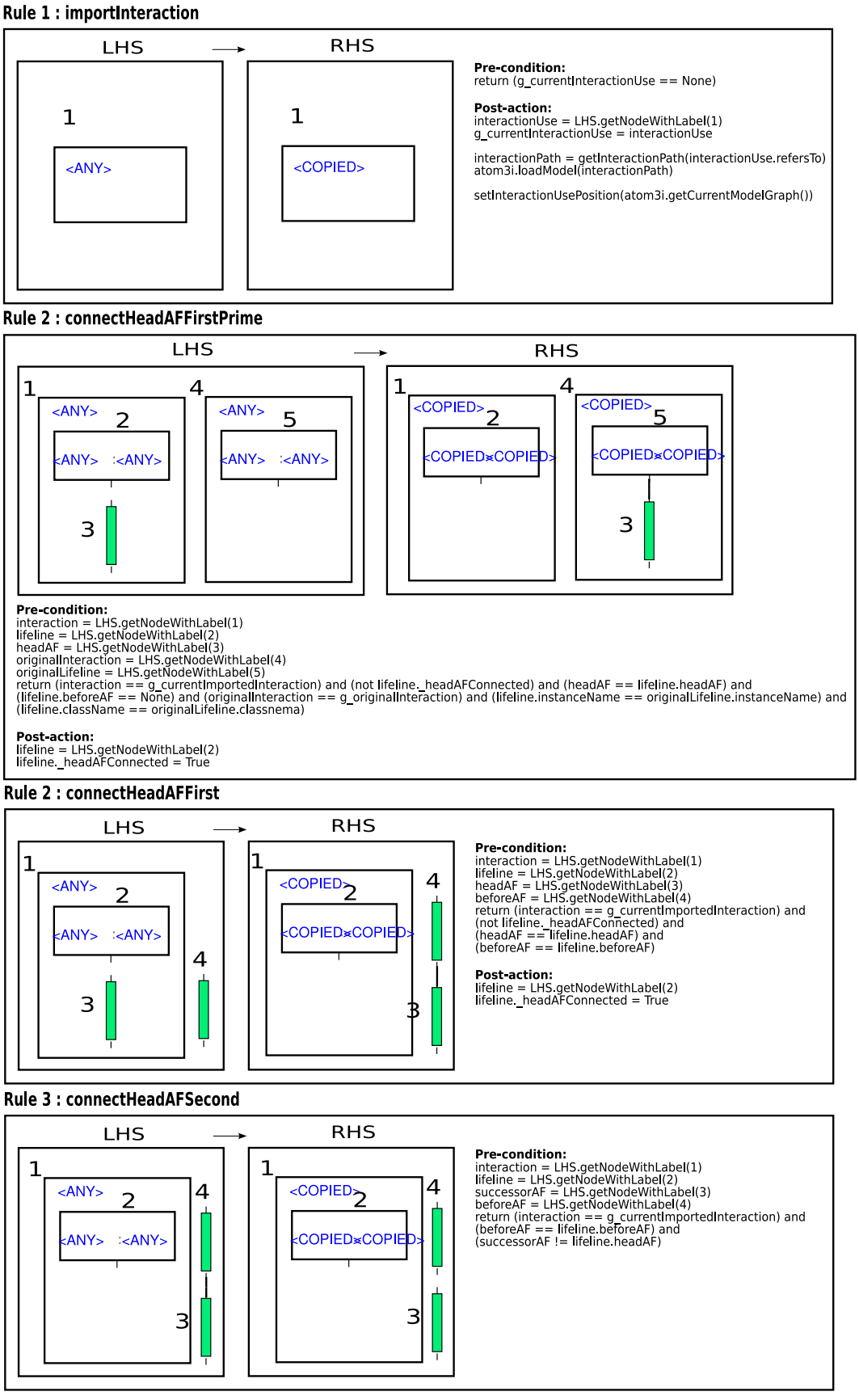

Figure 3.3: Graph Grammar rules in execution order in Phase Two - Part One

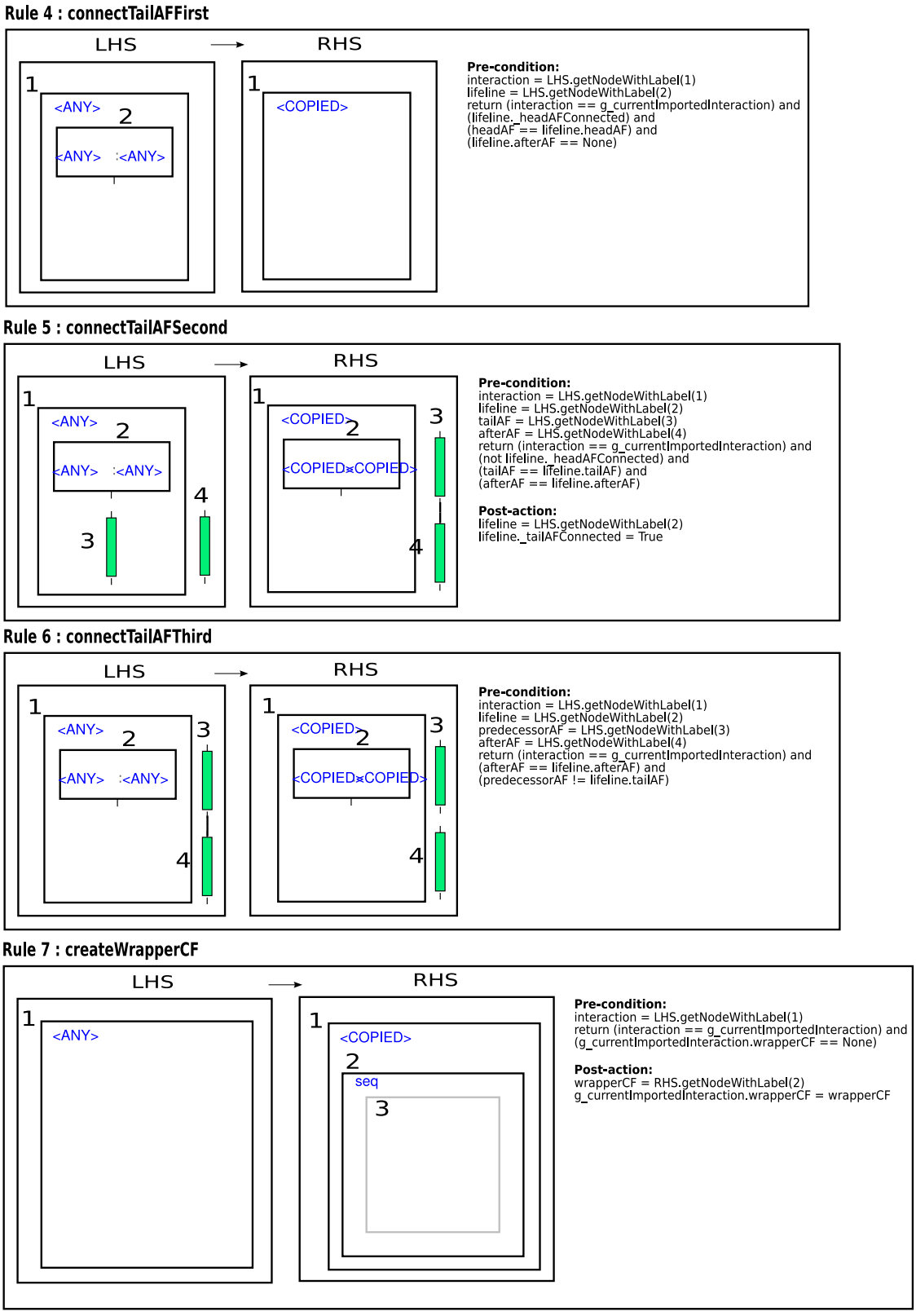

Figure 3.4: Graph Grammar rules in execution order in Phase Two - Part Two

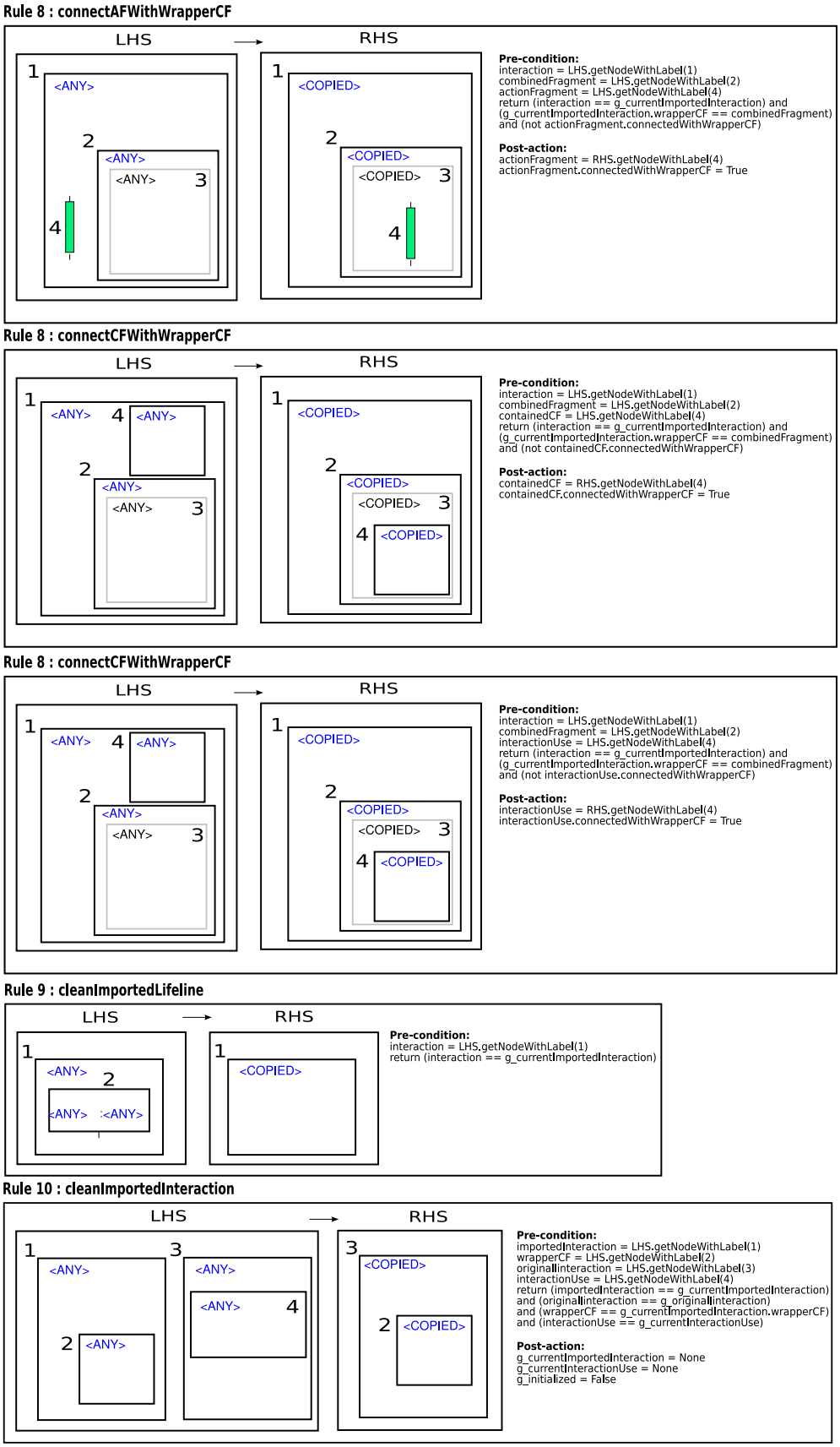

Figure 3.5: Graph Grammar rules in execution order in Phase Two - Part Three

Table 3.5: Settings of variables after applying Algorithm 3 on Figure 3.2

Lifeline	headAF	tailAF	beforeAF	afterAF
$a : A$	af_7	af_8	af_1	af_2
$b : B$	af_9	af_{10}	af_3	af_4

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. Second, it implies that the actual execution order of these rules is determined randomly and makes no difference. For example, the rules with order of 8.

The result of the transformation upon the example sequence diagram is shown in Figure 3.6. Note that number labels are added for action fragments which are not in the original models for ease of the illustration. Note how a special "seq" combined fragment which has only one operand is added by rule $createWrapperCF$ to enclose all elements imported from $Interaction\_2$. It is redundant after this phase and will be removed by the optimization described in Section 3.2.5.

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 $g\_currentInteractionUse$. The pre-condition of rule *importInteraction* checks if $g\_currentInteractionUse$ is null to start the new transformation of a matched interaction use element. The post-action of rule $cleanImportedInteraction$ sets $g\_currentInteractionUse$ to null to end the current iteration and to prepare for the next potential one.

3.2.4 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 follows 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 to transform a sequence diagram is depicted in Figure 3.7. 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.

There are sixteen rules in this phase which are shown in Figures 3.8, 3.9 and 3.10.

The list of rules, their execution order and short descriptions are given in Table 3.7.

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 components and states transformed from interaction operands and messages by the following rules. Rule *interactionOperand* is a general rule for all kinds of combined

Table 3.6: 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 (see Section 2.3.4).
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.

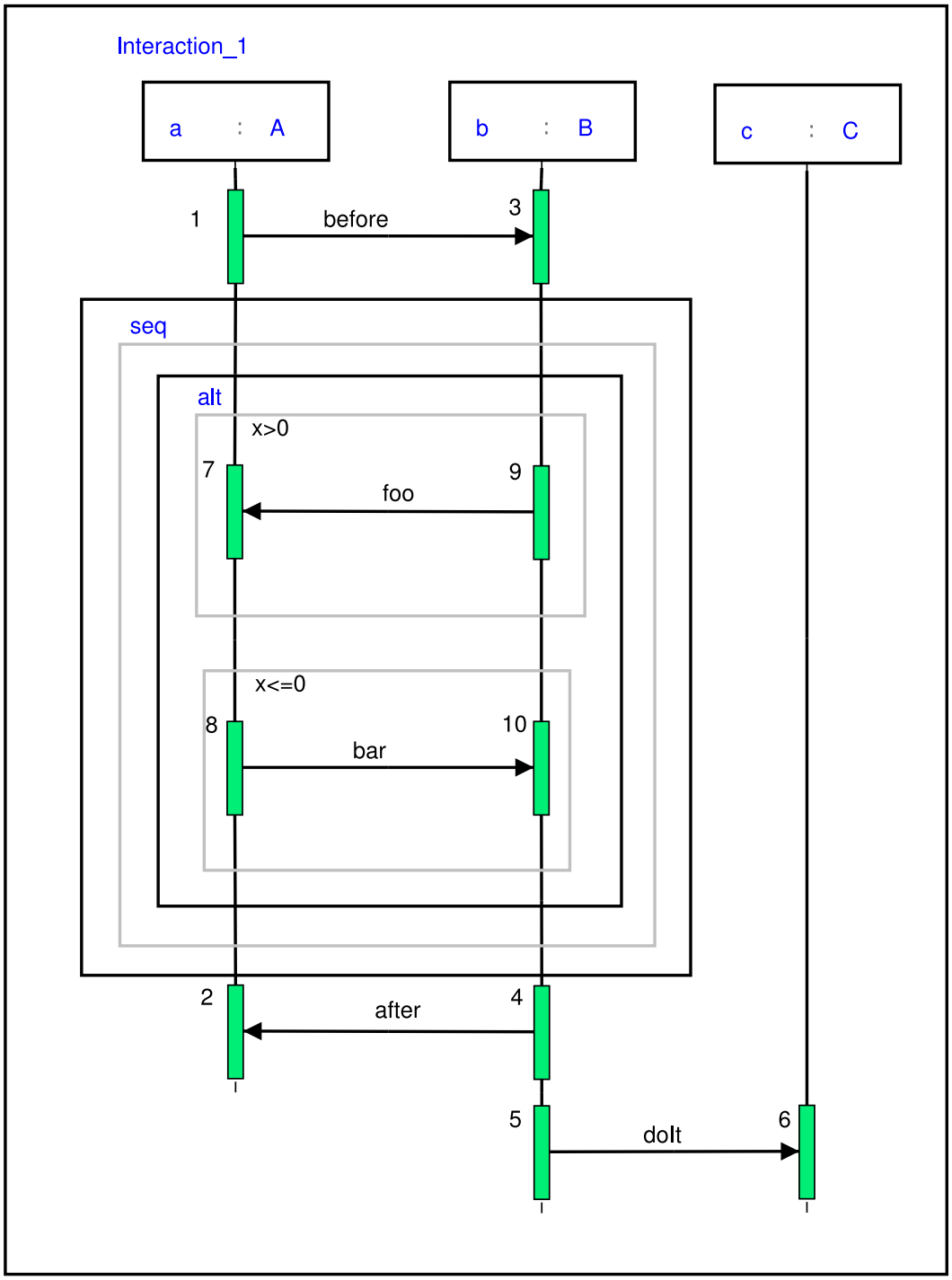

Figure 3.6: Transformation result of Phase Two

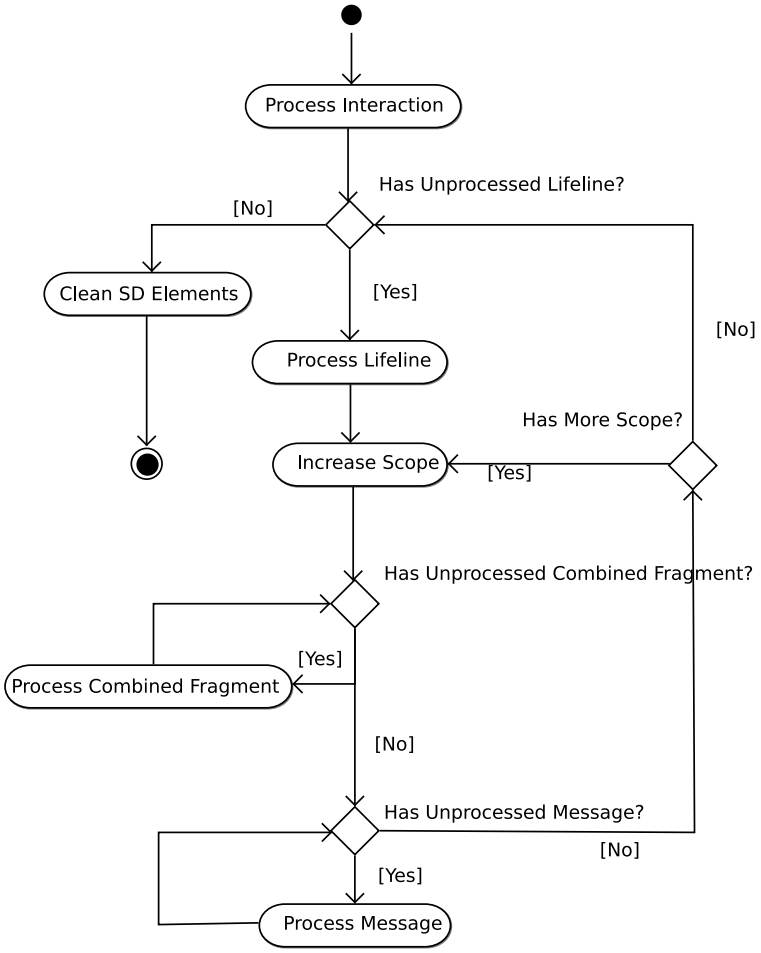

Figure 3.7: The strategy of Phase Three

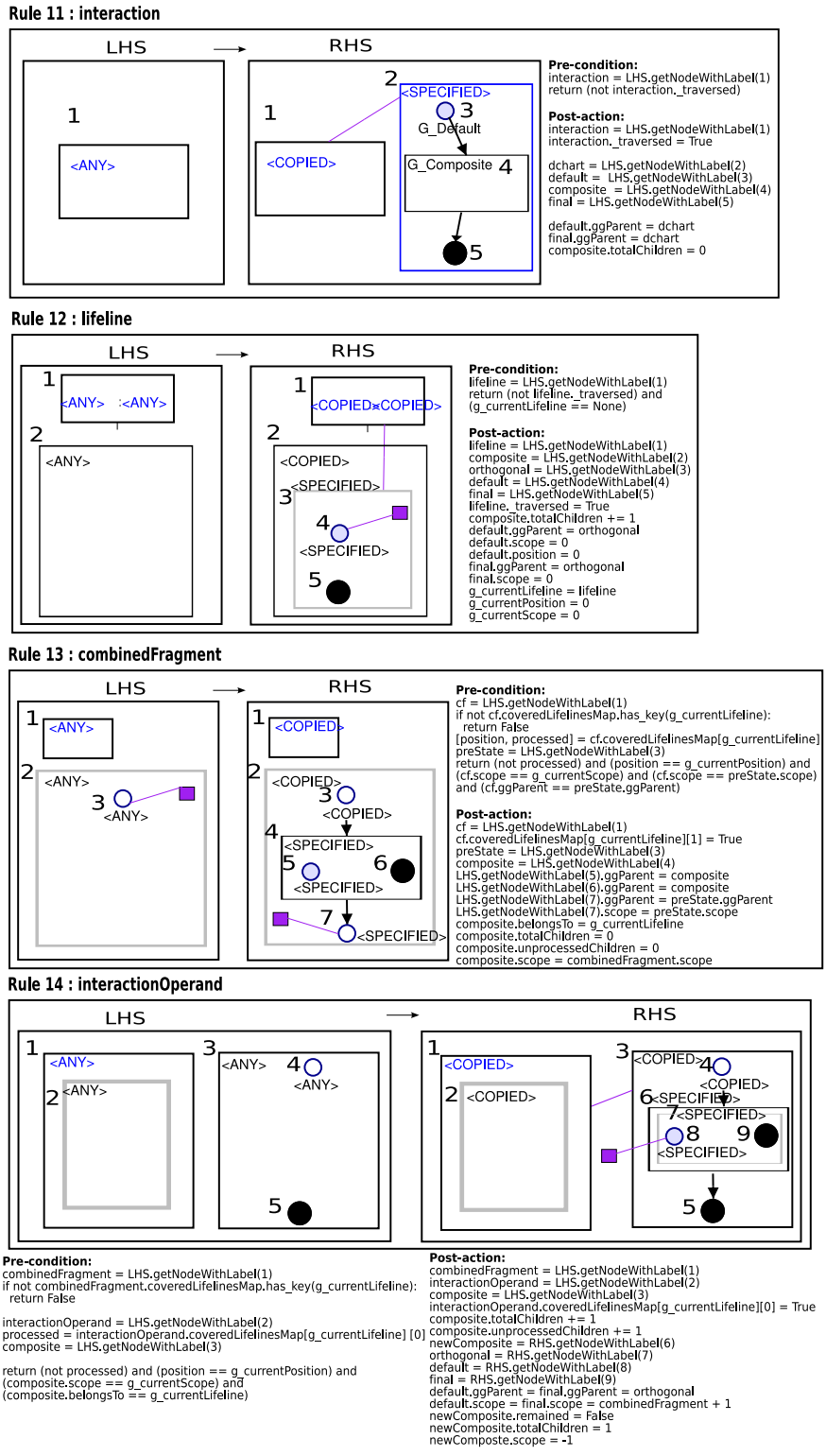

Figure 3.8: Graph Grammar rules in execution order in Phase Three - Part One

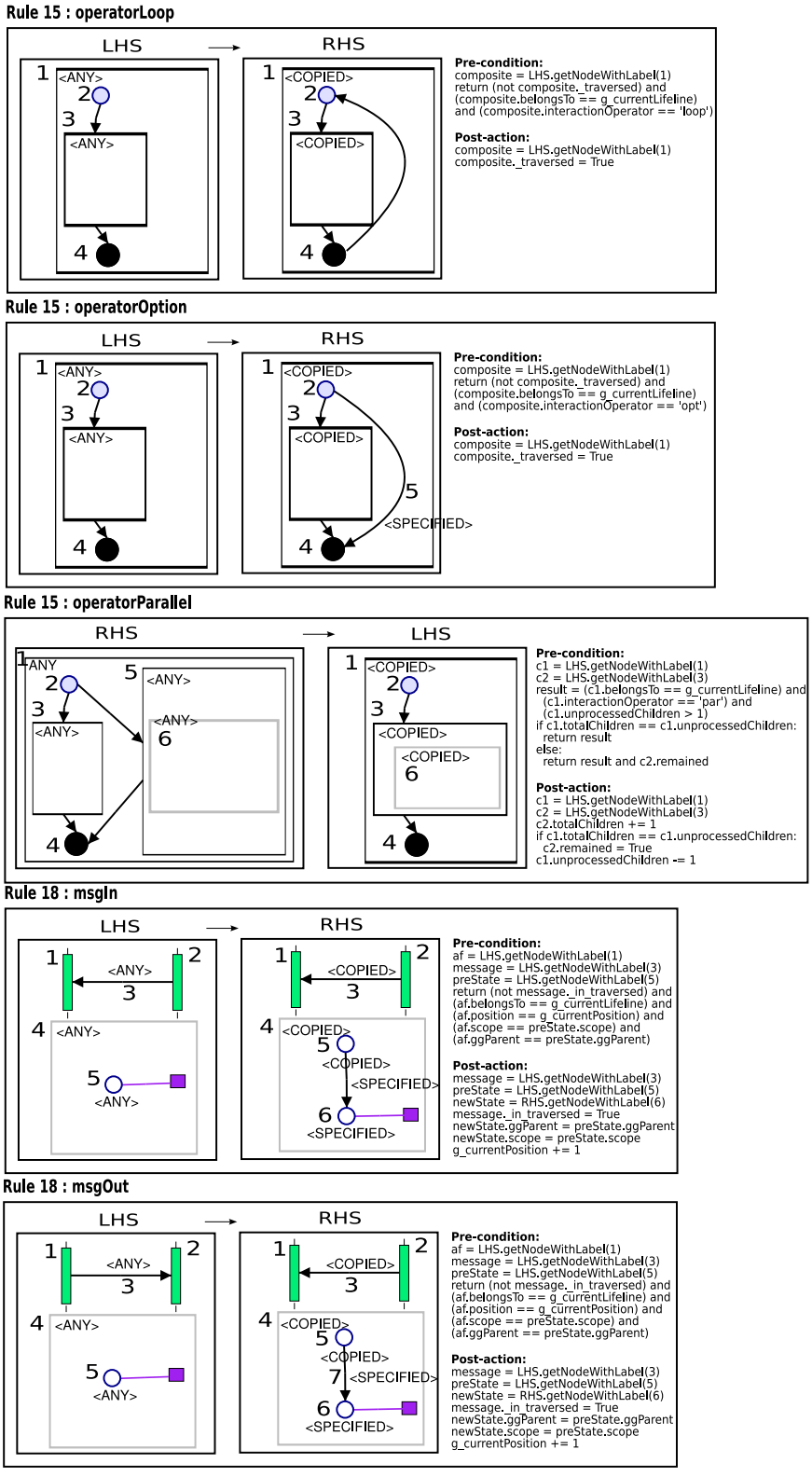

Figure 3.9: Graph Grammar rules in execution order in Phase Three - Part Two

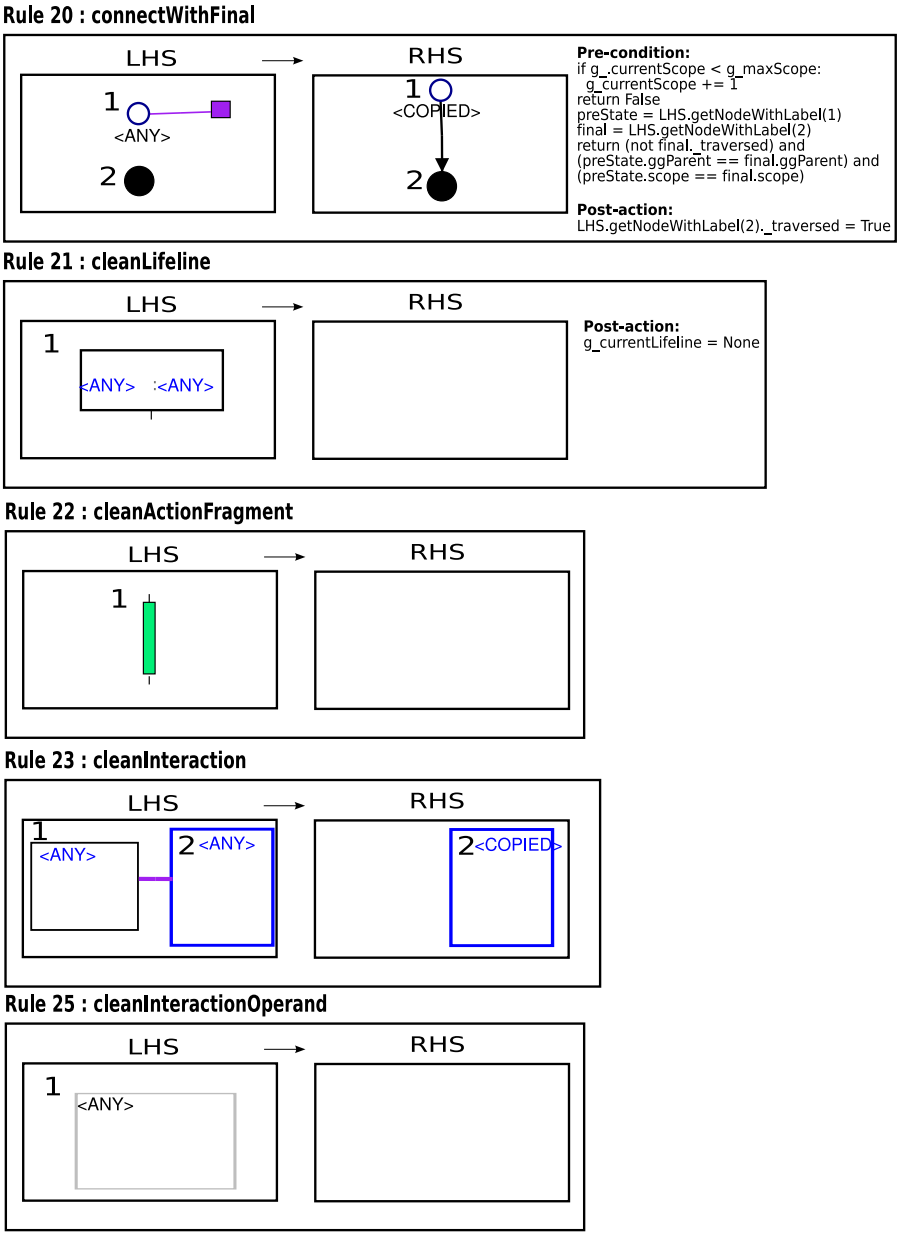

Figure 3.10: Graph Grammar rules in execution order in Phase Three - Part Three

Table 3.7: 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.

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 3.11. 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.

3.2.5 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.

There are sixteen rules in this phase which are shown in Figures 3.12 and 3.13.

Note that visual syntax of rule 34, 35, 36 and 37 are not shown in figures because they are very similar to those of 29, 30, 31 and 32. The only difference is that the outermost container element of the first set is a Composite component, and an Orthogonal component for the second

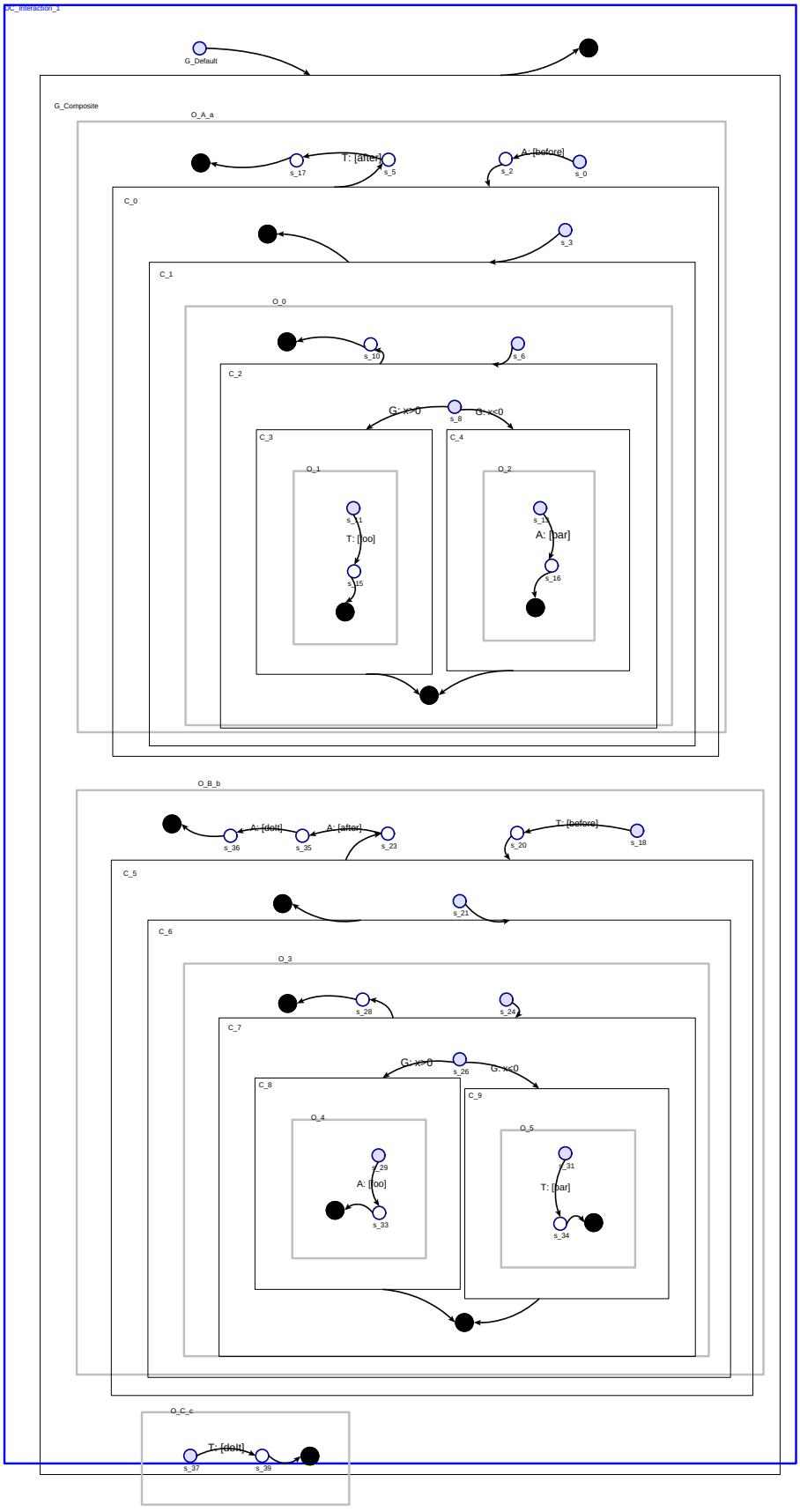

Figure 3.11: Transformation result of Phase Three

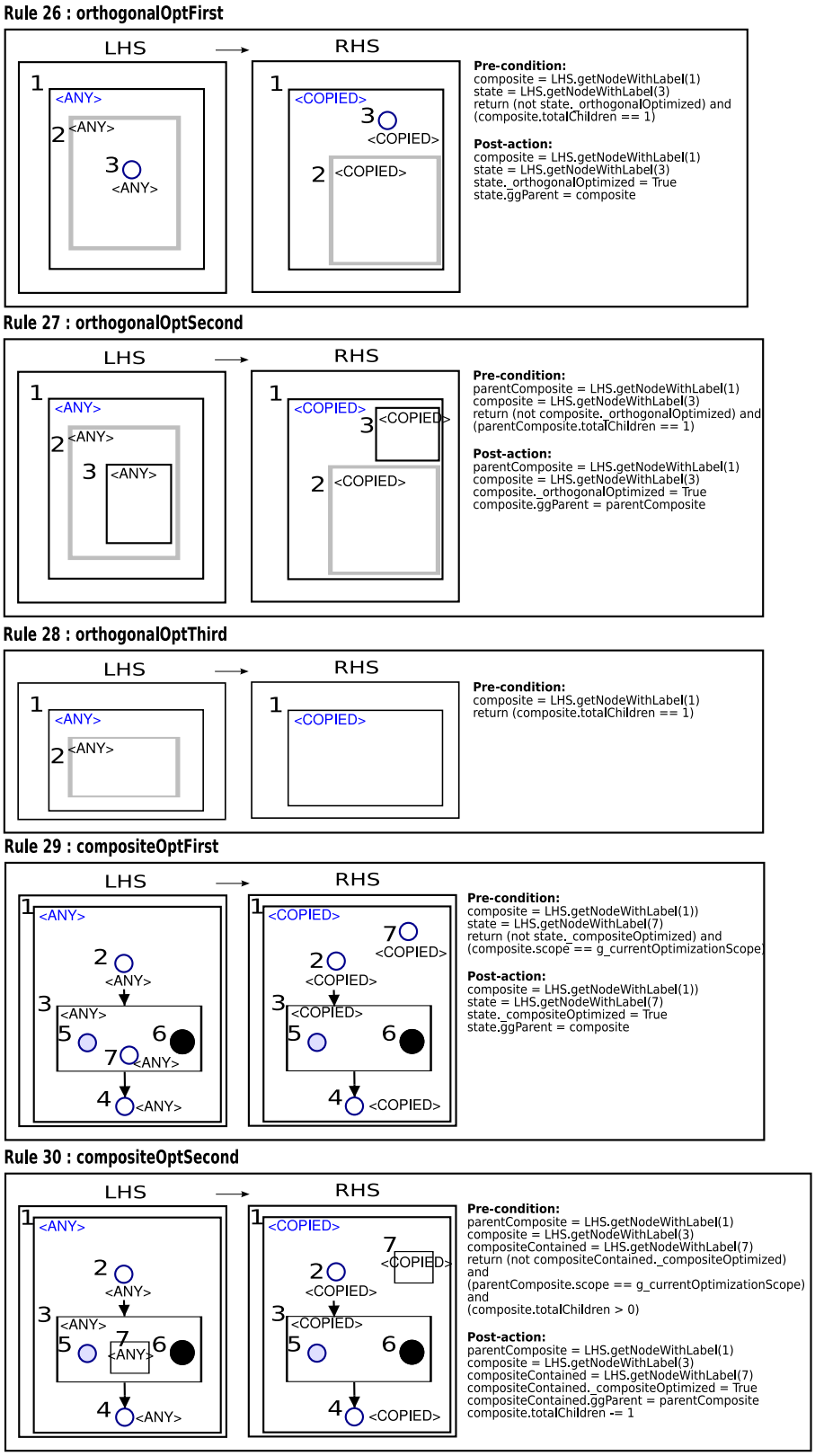

Figure 3.12: Graph Grammar rules in execution order in Phase Four - Part One

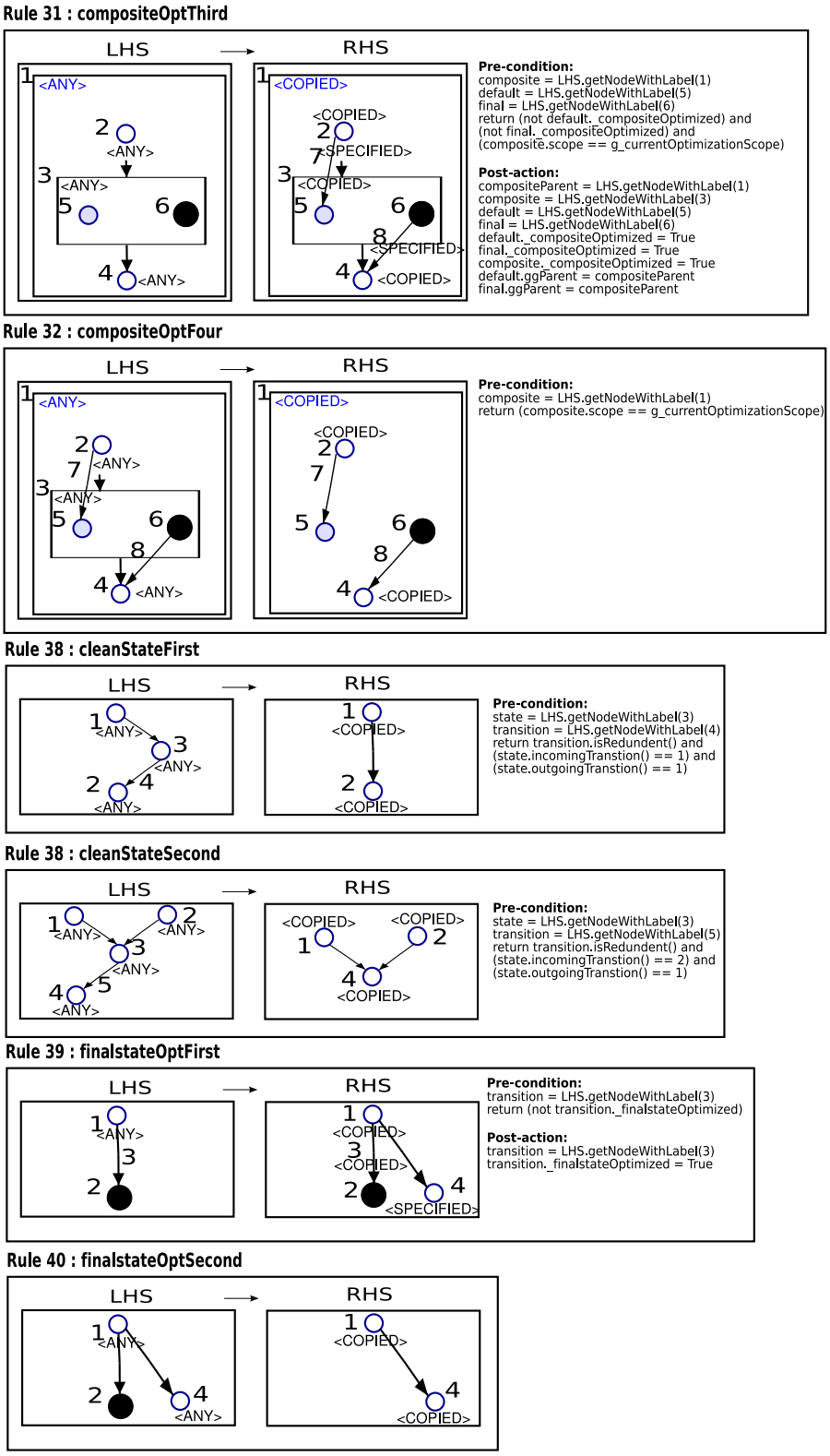

Figure 3.13: Graph Grammar rules in execution order in Phase Four - Part Two

set. The list of rules, their execution order and short descriptions are given in Table 3.8.

After these optimizations, the final statecharts transformed from the example of Figure 3.2 is shown in Figure 3.14.

3.3 Broadcasting Problem of Statecharts

As we use Harel's Statecharts as our target formalism of the transformation, generated models inherit the events broadcasting feature. That is, an event can be seen everywhere within a state machine at the same time. While sometimes this is simple and convenient, the broadcast property is not suited for modeling communication between objects.

For example, if there is another object $d : D$ in the example Sequence Diagram (shown in Figure 3.2) and $c : C$ sends the same message "before" to $d : D$ after $c : C$ receives message "doIt", then the generated Statechart will have an extra orthogonal component "O_D_d" which has a transition whose trigger is "before". The problem happens when the event "before" is triggered by the first transition in "O_A_a". Both transitions with the trigger event "before" in "O_B_b" and "O_D_d" are triggered, which is not a desired effect.

In the UML Statecharts, events are not broadcast globally. Broadcasting happens only within an object. Events may be sent (multicast) to an identified set of objects through explicit channels between them.

In this thesis, we took a simpler way to solve this problem. We narrow out the broadcasting by giving events (transformed from messages in Sequence Diagrams) globally unique names. Applied to the same example described above, message "before" sending from $a : A$ to $b : B$ will be transformed into event "before_0" and message "before" sending from $c : C$ to $d : D$ will be transformed into event "before_1".

3.4 Behavior Trace Comparison

In this section, we give an informal comparison of behavior traces between a Sequence Diagram model (shown in Figure 3.2) and its transformed Statechart model (shown in Figure 3.14).

The Sequence Diagram model $Interaction\_1$ (shown on the left of Figure 3.2) has two possible complete traces because of the use of an "alternative" combined fragment (containing two operands) in the reference interaction $Interaction\_2$ (shown on the right of Figure 3.2). The same thing is to the Statechart model $DC\_Interaction\_1$.

We illustrate the traces as follows:

1. In $Interaction\_1$, lifeline $a : A$ sends message "before" to $b : B$. Correspondingly, in $DC\_Interaction\_1$, after the first transition from "G_Default" to "G_Composite", event "before" is generated with the transition $T_{s\_0 \rightarrow s\_8}$ in orthogonal component "O_A_a" which immediately triggers the transition $T_{s\_18 \rightarrow s\_26}$ in "O_B_b";
2. In $Interaction\_1$, a choice is made based on the value of variable "x". That is, if $x > 0$, then $b : B$ sends message "foo" to $a : A$; otherwise, $a : A$ sends message "bar" to $b : B$. This corresponds to a series of transitions in $DC\_Interaction\_1$: if $x > 0$, two transitions $T_{s\_8 \rightarrow s\_18}$ and $T_{s\_26 \rightarrow s\_31}$ are triggered in parallel followed by transitions $T_{s\_18 \rightarrow s\_5}$ and $T_{s\_31 \rightarrow s\_28}$; if $x \leq 0$, two transitions $T_{s\_8 \rightarrow s\_11}$ and $T_{s\_26 \rightarrow s\_29}$ are triggered in parallel followed by transitions $T_{s\_11 \rightarrow s\_5}$ and $T_{s\_29 \rightarrow s\_28}$;
3. In $Interaction\_1$, lifeline $b : B$ sends message "after" to $a : A$. Correspondingly, in $DC\_Interaction\_1$, event "after" is generated with the transition $T_{s\_28 \rightarrow s\_35}$ in orthogonal

Table 3.8: 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).

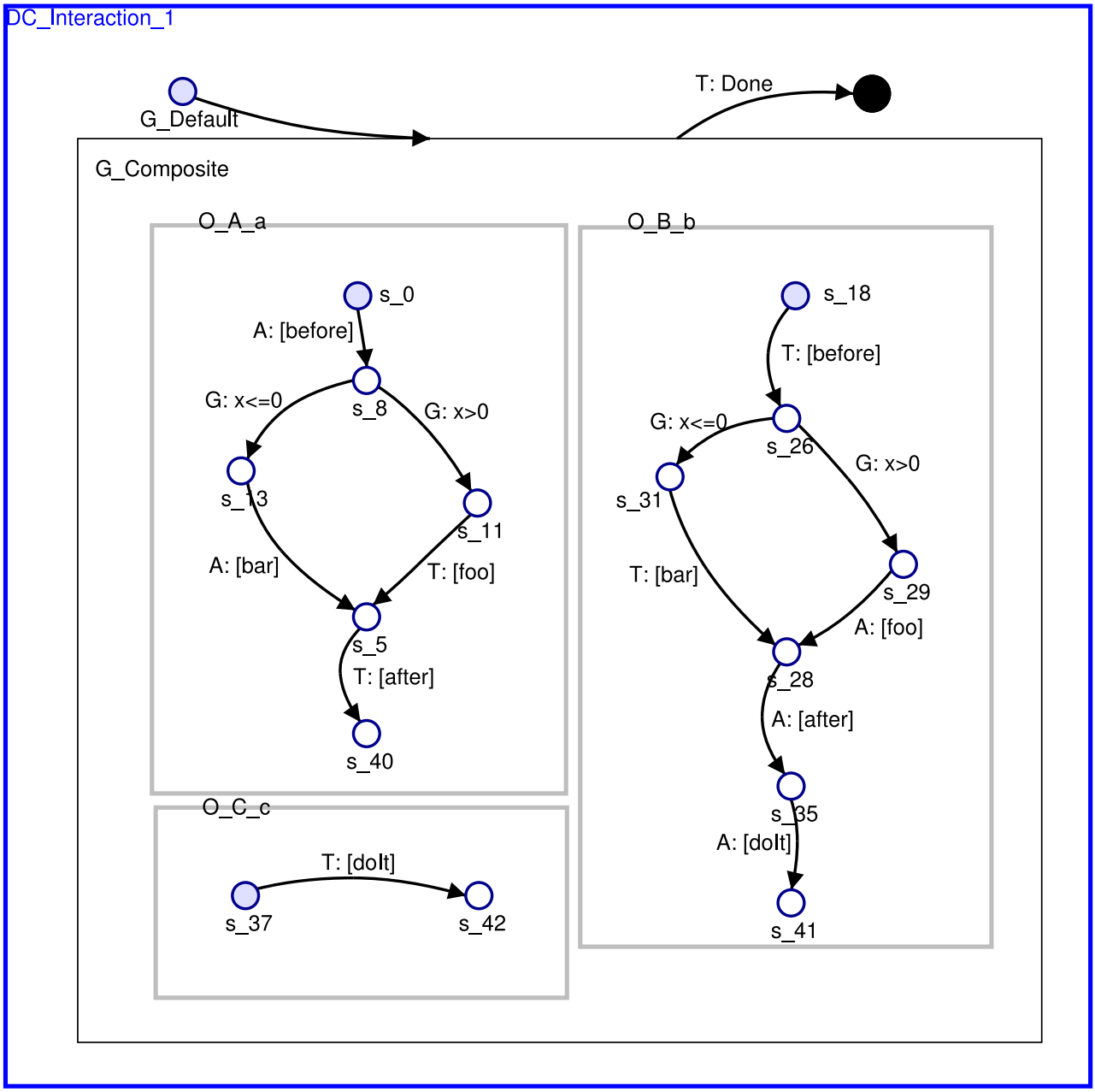

Figure 3.14: The final generated Statechart for the example of Figure 3.2

component "O_B_b" which immediately triggers the transition $T_{s\_5 \rightarrow s\_40}$ in "O_A_a";

4. Finally, in $Interaction\_1$, lifeline $b : B$ sends message "bar" to $c : C$. This corresponds to the generation of event "bar" with the transition $T_{s\_35 \rightarrow s\_41}$ in orthogonal component "O_B_b" which immediately triggers the transition $T_{s\_5 \rightarrow s\_40}$ in "O_C_c", in $DC\_Interaction\_1$.

3.5 Transformation Application on a Simple Sequence Diagram Model

In this section, we give a detailed description about the application of the graph grammars we defined in previous sections to the Sequence Diagram model shown in Figure 3.2. We only explain the key steps the transformation process. The complete list of steps is given in Appendix A.

The results of Step 1 with execution of procedures *setPosition* (see Algorithm 1), *setScope* (see Algorithm 2) and *initiateGlobals*) are listed in Tables 3.9 and 3.10.

Table 3.9: Transformation result of Rule *initInteraction* - global variables assignment

Name	**Value**
g_initialized	True
g_maxScope	0
g_topLevelCombinedFragments	[]
g_transitionCount	0
g_stateCount	0
g_currentLifeline	None
g_currentPosition	0
g_currentScope	0
g_currentOptimizationScope	0
g_currentInteractionUse	None
g_currentImportedInteraction	None
g_originalInteraction	Interaction_1

Steps 2 to 10 are shown in Figure 3.15. In Step 2, *g_currentInteractionUse* is set and the the procedure *setInteractionUsePosition* (see Algorithm 3) is executed which generates results shown in Table 3.4.

After Steps 11 to 13 clean imported lifelines and interaction and reset some global variables, the transformed model is shown in Figure 3.6. Step 14 executes Rule *initInteraction* again and reset variables. Now, *g_maxScope* is set to 2 and some other local variables are set as shown in Table 3.11.

Figure 3.16 shows Steps 15 to 31 which complete the transformation from lifeline $a : A$ to orthogonal component "O_A_a".

Steps 15 to 18 can be easily understood from Figure 3.16. Step 19 adds composite component

Table 3.10: Local variables setting of ActionFragments - Round One

Element	**belongsTo**	**position**	**scope**
af_1	$lifeline_{a:A}$	0	0
af_2	$lifeline_{a:A}$	1	0
af_3	$lifeline_{b:B}$	0	0
af_4	$lifeline_{b:B}$	1	0
af_5	$lifeline_{b:B}$	2	0
af_6	$lifeline_{c:C}$	0	0

Table 3.11: Local variables setting of ActionFragments - Round Two

Element	**belongsTo**	**position**	**scope**
af_1	$lifeline_{a:A}$	0	0
af_2	$lifeline_{a:A}$	3	0
af_3	$lifeline_{b:B}$	0	0
af_4	$lifeline_{b:B}$	3	0
af_5	$lifeline_{b:B}$	4	0
af_6	$lifeline_{c:C}$	0	0
af_7	$lifeline_{a:A}$	1	2
af_8	$lifeline_{a:A}$	2	2
af_9	$lifeline_{b:B}$	1	2
af_{10}	$lifeline_{b:B}$	2	2

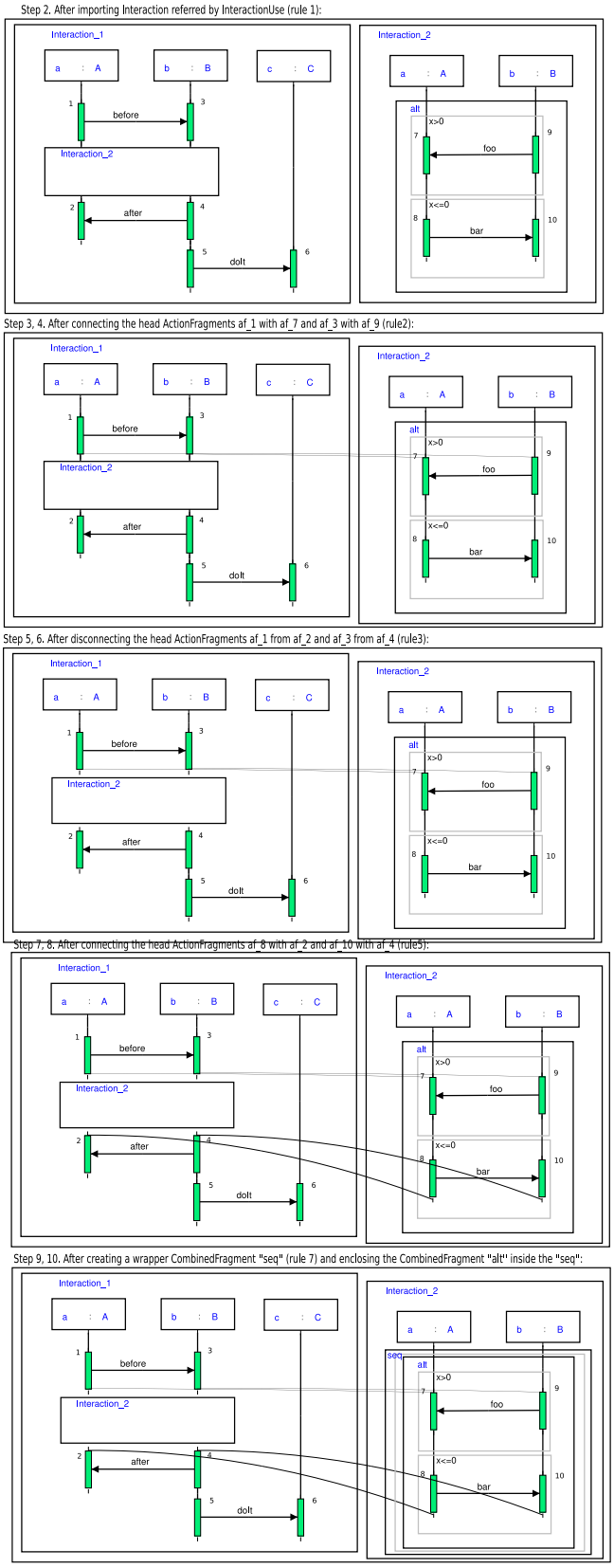

Figure 3.15: Steps 2 to 10 of the transformation from Sequence Diagrams to Statecharts

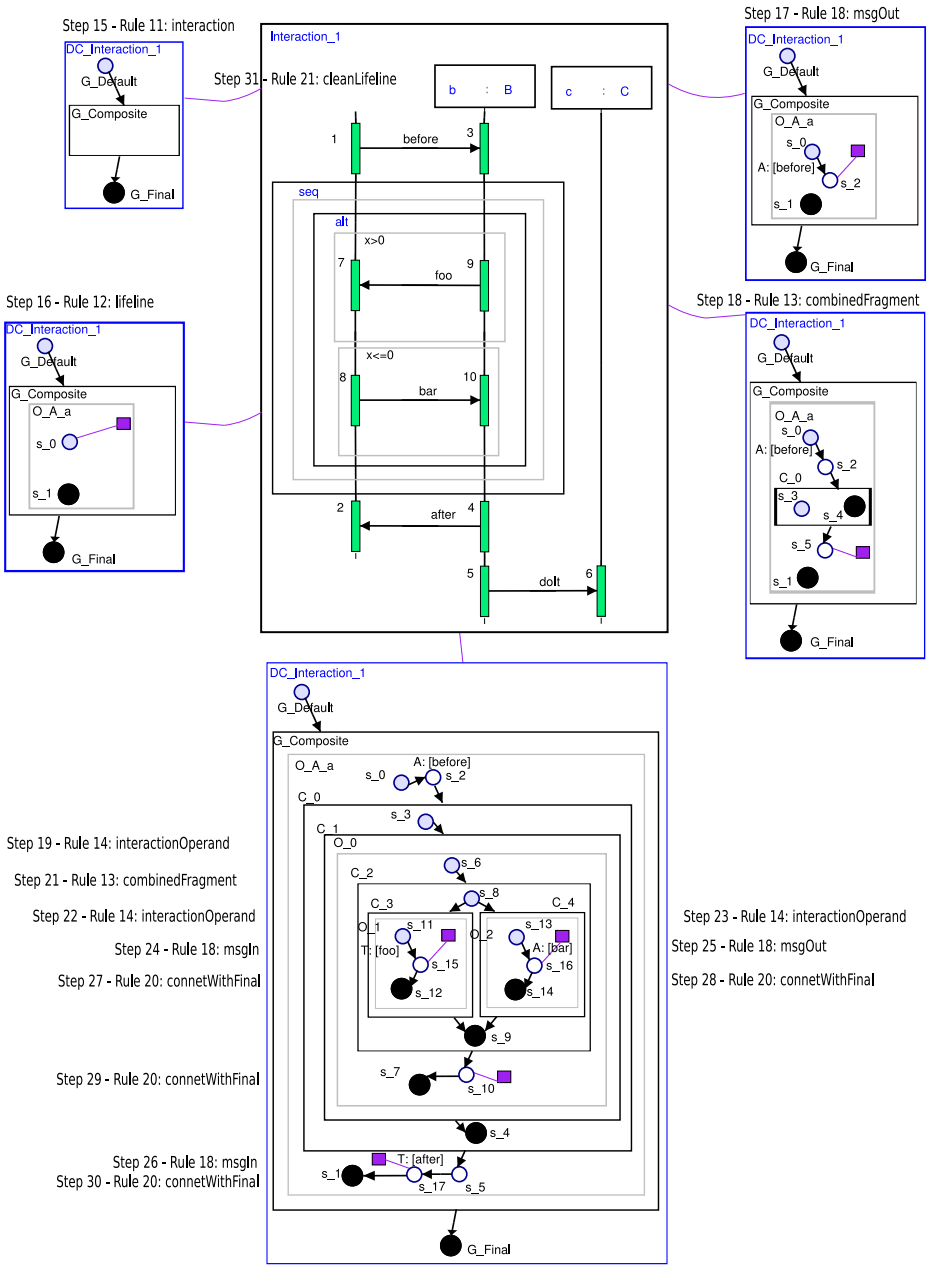

Figure 3.16: Steps 15 to 30 of the transformation from Sequence Diagrams to Statecharts

C_1, orthogonal component O_0, states s_6 and s_7, and transitions $T_{s\_3 \rightarrow C\_1}$ and $T_{C\_1 \rightarrow s\_4}$. Step 20 (Rule *setScope*) increases *g_currentScope* by 1. Step 21 adds composite C_2, states s_8 and s_9 and s_10, and transitions $T_{s\_6 \rightarrow C\_2}$ and $T_{C\_2 \rightarrow s\_10}$. Step 22 adds composite component C_3, orthogonal component O_1, states s_11 and s_12, and transitions $T_{s\_8 \rightarrow C\_3}$ and $T_{C\_3 \rightarrow s\_9}$. Step 23 adds composite component C_4, orthogonal component O_2, states s_13 and s_14, and transitions $T_{s\_8 \rightarrow C\_4}$ and $T_{C\_4 \rightarrow s\_9}$. Step 24 adds state s_15 and a transition $T_{s\_11 \rightarrow s\_15}$. Step 25 adds state s_16 and a transition $T_{s\_13 \rightarrow s\_16}$. Step 26 adds state s_17 and a transition $T_{s\_5 \rightarrow s\_17}$. Step 27 adds a transition $T_{s\_15 \rightarrow s\_12}$. Step 28 adds a transition $T_{s\_16 \rightarrow s\_14}$. Step 29 adds a transition $T_{s\_10 \rightarrow s\_7}$. Step 30 adds a transition $T_{s\_17 \rightarrow s\_1}$. Step 31 removes lifeline $a : A$.

Steps 32 to 53 are very similar to those shown in Figure 3.16 as they transform lifeline $b : B$ to orthogonal component "O_B_b" and lifeline $c : C$ to orthogonal component "O_C_c" respectively. From Step 54 to Step 69, all remained Sequence Diagrams elements are removed. We get the Statecharts model shown in Figure 3.11.

From Step 70 until the end, the transformed model is further optimized to reduce unnecessary states, composite and orthogonal components. Although the amount of steps is quite large (over one hundred), the number of corresponding rules is not that large and most of them are very simple. Steps 70 to 89 reconnect state or composite nodes of the redundant orthogonal components with these orthogonals parents. Then in Steps 90 to 95, those orthogonal components are safely removed. Steps 96 to 152 do similar things but result in the removing of redundant composite components. The remaining steps are even simpler which remove redundant states. Finally, the transformation is complete and the generated Statechart is shown in Figure 3.14.

3.6 Requirements Generation

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 [Li00] 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.

Another way to help fill the gap between Use Cases and Sequence Diagrams is the inverse of the previous mapping. As mentioned in Section 1.4, 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 the earliest stage.

In the following sections, we describe the algorithms of the textual requirements generation from Sequence Diagrams models and shows its application by means of an example.

3.6.1 Algorithms

The algorithm of the main procedure is listed in Algorithm 4. The first step (shown in Algorithm 5) is to initialize variables of each lifeline and the action fragments along it. These initialized variables are used in the next steps. The second step (shown in Algorithm 6) is to set proper scope information of combined fragments, interaction operands, interaction uses, and action fragments. The algorithm is similar to Algorithm 2 in section 3.2.2. The third step (shown in Algorithm 7) is to create a list which contains all message sends. Each element of

the list is a tuple which stores the information of the source, the target, and the message itself. The list is sorted by time sequence of the messages, visually from the top to the bottom of a diagram. The fourth step (shown in Algorithm 8) is to insert interaction uses into the list created in the previous step. The proper insertion position is decided by the visual position where an interaction use appears in the diagram. That is, an interaction use is inserted either to the first position of the list because it is the first element on the time axis, or after the message send which is the closest one before the interaction use. The last step (shown in Algorithm 9) is to iterate through the list to emit text description of each element, either a message send or an interaction use.

Algorithm 4 genReq(Graph graph)

```
lifelines = initializeLifelines(graph)
setScopes(graph)
messages = initializeMessages(lifelines)
insertInteractionUses(graph, messages)
emitText(graph, messages)
```

Algorithm 5 initializeLifelines(Graph graph)

```
sort all lifelines in graph in order of their geometric positions
for all lifeline in lifelines do
  lifeline.actionFragments = [ ]
  while hasNextActionFragment(lifeline) do
    actionFragment = getNextActionFragment(lifeline)
    actionFragment.belongsTo = lifeline
    actionFragment.scope = 0
    actionFragment.scopeHasSet = False
    lifeline.actionFragments.append(actionFragment)
  end while
end for
return lifelines
```

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 any 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 include 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.

Algorithm 6 setScopes(Graph graph)

```
interaction = getInteraction(graph)
for all combinedFragment in interaction do
  setScopeHelper(combinedFragment, 0)
end for
for all interactionUse in interaction do
  interactionUse.scope = 0
end for
## helper function for setting scope recursively
setScopeHelper(CombinedFragment combinedFragment, int scope):
combinedFragment.scope = scope
combinedFragment.emitted = False
combinedFragment.operandsNum = 0
combinedFragment.emittedOperandsNum = 0
for all interactionOperand contained in combinedFragment do
  interactionOperand.parentContainer = combinedFragment
  interactionOperand.emitted = False
  combinedFragment.operandsNum += 1
end for
for all containedEntity contained in interactionOperand do
  containedEntity.parentContainer = interactionOperand
  if isinstance(containedEntity, ActionFragment) and not containedEntity.scopeHasSet
  then
    containedEntity.scope = scope + 1
    containedEntity.scopeHasSet = True
  end if
  if isinstance(containedEntity, InteractionUse) and not containedEntity.scopeHasSet then
    containedEntity.scope = scope + 1
    containedEntity.scopeHasSet = True
  end if
  if isinstance(containedEntity, CombinedFragment) then
    setScopeHelper(combinedFragment, scope+1)
  end if
end for
```

Algorithm 7 initializeMessages(List lifelines)

```
messages = [ ]
for all lifeline in lifelines do
  for all actionFragment in lifeline.actionFragments do
    source = actionFragment
    message = getOutLink(actionFragment)
    target = getOutNode(message)
    messages.append((source, target, message))
  end for
end for
sort messages in the order of time when they happen
return messages
```

Algorithm 8 insertInteractionUses(Graph graph, List messages)

```
for all interactionUse contained in graph do
  interactionUse.closestAF = None
  for all lifeline covered by interactionUse do
    while hasNextActionFragment(lifeline) do
      actionFragment = getNextActionFragment(lifeline)
      if actionFragment is ahead of interactionUse then
        if interactionUse.closestAF not None then
          if actionFragment is ahead of interactionUse.closestAF then
            interactionUse.closestAF = actionFragment
          end if
        else
          interactionUse.closestAF = actionFragment
        end if
      end if
    end while
  end for
  if interactionUse.closestAF not None then
    index = 0
    for all (source, target, message) in messages do
      index += 1
      if source == interactionUse.closestAF or target == interactionUse.closestAF then
        messages.insert(index, interactionUse)
        break
      end if
    end for
  else
    messages.insert(0, interactionUse)
  end if
end for
```

Algorithm 9 emitText(Graph graph, List messages)

```
generate interaction name text
generate actors list text
for all message in messages do
  if isinstance(message, InteractionUse) then
    generate message text as an InteractionUse
  else
    (s, t, m) = message
    if source.scope > 0 then
      interactionOperand = source.parentContainer
      combinedFragment = interactionOperand.parentContainer
      if not interactionOperand.emitted then
        interactionOperator = combinedFragment.interactionOperator
        generate output w.r.t. the interactionOperator type
        interactionOperand.emitted = True
        combinedFragment.emittedOperandsNum += 1
      end if
    end if
    generate message text
  end if
end for
```

"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:

$$\text{“}Interaction\text{”} + refersTo + \text{“}isimportedhere\text{”}$$

3.6.2 Example

We apply Algorithm 4 to the simple Sequence Diagrams model (shown in Figure 3.2) 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
```

4

Case Study: Model-Driven Dependability Analysis on An Elevator Control System

In this chapter, we demonstrate the usefulness of our model-driven approach in the content of scenario-based requirements engineering, by means of an elevator control system case study. In particular, the case study shows how the model-driven approach helps developers to model and analyze the dependability of use cases and to discover more reliable and safe ways of designing the interactions with the system and the environment.

The demonstration described here extends our previous work [MSKV06] by automating the process of mapping use cases from scenario-based models (i.e., Sequence Diagrams) to state-based models (i.e., Statecharts).

Section 4.1 provides background information on dependability, use cases, and the exceptional use cases approach in [SMKD05]. Section 4.2 describes our model-driven process for assessing and refining use cases. Section 4.3 presents our probabilistic statecharts formalism used for dependability analysis. Section 4.4 illustrates our proposed process by means of an elevator control system case study.

4.1 Background

4.1.1 Dependability

Dependability [LAK92] is that property of a computer system such that reliance can justifiably be placed on the service it delivers. It involves satisfying several requirements: availability, reliability, safety, maintainability, confidentiality, and integrity. The dependability requirement varies with the target application, since a constraint can be essential for one environment and not so much for others. In the following, we focus on the *reliability* and *safety* attributes of dependability. The **reliability** of a system measures its aptitude to provide service and remain operating as long as required [GM02]. The **safety** of a system is determined by the lack of catastrophic failures [GM02].

Fault tolerance is a means of achieving system dependability. As defined in [ALR01], fault tolerance includes error detection and system recovery. At the use case level, error detection involves detection of exceptional situations by means of secondary actors such as sensors and time-outs. Recovery at the use case level involves describing the interactions with the environment that are needed to continue to deliver the current service, or to offer a degraded service, or to take actions that prevent a catastrophe. The former two recovery actions increase reliability, whereas the latter ensures safety.

4.1.2 Use Cases

Use cases are a widely used formalism for discovering and recording behavioral requirements of software systems [Lar02]. A use case describes, without revealing the details of the system's internal workings, the system's responsibilities and its interactions with its environment as it performs work in serving one or more requests that, if successfully completed, satisfy a goal of a particular stakeholder. The external entities in the environment that interact with the system are called *actors*.

Use cases are stories of actors using a system to *meet goals*. The actor that wants to achieve the goal is referred to as the *primary actor*. Entities that the system needs to fulfill the goal are called *secondary actors*. Secondary actors include software or hardware that is out of our control. The system, on the other hand, is the software that we are developing and which is under our control.

4.1.3 Exceptions and Handlers in Use Cases

In [SMKD05], an approach was proposed that extends traditional use case driven requirements elicitation, leading the analyst to focus on all possible exceptional situations that can interrupt normal system interaction.

An exception occurrence endangers the completion of the actor's goal, suspending the normal interaction temporarily or for good. To guarantee reliable service or ensure safety, special interaction with the environment might be necessary. These handling actions can be described in a *handler use case*. That way, from the very beginning, exceptional interaction and behavior is clearly identified and separated from the normal behavior of the system. Similar to standard use cases, handler use cases are reusable. Handlers can be defined for handlers in order to specify actions to be taken when an exception is raised in a handler itself.

4.2 Model-Driven Dependability Analysis of Use Cases

In [MSKV06], we proposed a model-driven approach for assessing and refining use cases to ensure that the specified functionality meets the dependability requirements of the system as defined by the stakeholders.

Now we extends our previous work by automating the process of mapping use cases from scenario-based models (i.e., Sequence Diagrams) to state-based models (i.e., Statecharts).

For the purpose of analysis, we introduce probabilities in use cases. The value associated with each interaction step represents the probability with which the step succeeds. If we assume reliable communication and a perfect software (which we must at the requirements level), the success and failure of each interaction depends on the quality of the hardware device, e.g. motor, sensor, etc. The reliability of each hardware component can be obtained from the manufacturer. If the secondary actor is a software system, its reliability is also either known or must be determined statistically.

Our proposed process is illustrated in Figure 4.1. First, the analyst starts off with standard use case-driven requirements elicitation (see step 1). Using the exceptional use case approach described in[SMKD05] the analyst discovers exceptional situations, adds detection hardware to the system if needed, and refines the use cases (see step 2). Then, each use case step that represents an interaction with a secondary actor is annotated with a probability value that specifies the chances of success of the interaction (see step 3). Additionally, each interaction step is annotated with a safety tag if the failure of that step threatens the safety of the system.

Next, each use case is modeled in Sequence Diagrams (see step 4). Then, these models are automatically mapped to Statecharts and manually annotated with probabilities in DA-Charts (see step 5). The reasons why we do not give probabilities in Sequence Diagrams are discussed in Section 4.4.4. DA-Charts and the mapping process are described in Section 4.3. The DA-Charts are then mathematically analyzed by our dependability assessment tool (see step 6) and a report is produced.

The assessment report allows the analyst to decide if the current system specification achieves the desired reliability and safety. If not, several options can be investigated. It is possible to increase the reliability of secondary actors by, for instance, buying more reliable hardware components, or employing redundant hardware and voting techniques. Alternatively, the use cases have to be revisited and refined. First, the system must be capable of detecting the exceptional situation. This might require the use of time-outs, or even the addition of detection hardware to the system. Then, handler use cases must be defined that compensate for the failure of the actor, or bring the system to a safe halt. The analyst can perform the refinements in the annotated use cases or on the sequence diagrams.

After the changes, the effects on the system reliability and safety are determined by re-running the probabilistic analysis. The refinement process is repeated until the stakeholders are satisfied.

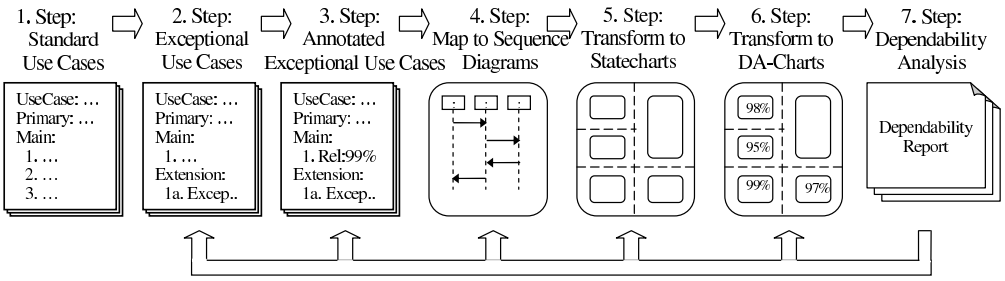

Figure 4.1: Model-Driven Process for Assessment and Refinement of Use Cases

4.2.1 Elevator System

We demonstrate our approach by applying it to an elevator control system case study. An elevator system is a hard real-time application requiring high levels of dependability.

For the sake of simplicity, there is only one elevator cabin that travels between the floors. The job of the development team is to decide on the required hardware, and to implement the elevator control software that processes the user requests and coordinates the different hardware devices. Initially, only "mandatory" elevator hardware has been added to the system: a motor to go up, go down or stop; a cabin door that opens and closes; floor sensors that detect when the cabin is approaching a floor; two buttons on each floor to call the elevator; and a series of buttons inside the elevator cabin.

Standard use case-driven requirements elicitation applied to the elevator control system results in the use case model shown in Figure 4.2. In the elevator system there is initially only one primary actor, the *User*. A user has only one goal with the system: to take the elevator to go to a destination floor. The primary actor *(User)* is the one that initiates the TakeLift use case. All secondary actors (the *Door*, the *Motor*, the *Exterior* and *Interior Floor Buttons*, as well as the *Floor Sensors*) that collaborate to provide the user goal are also depicted. For simplicity,

we only discuss the subfunction level use case *ElevatorArrival* (shown in Figure 4.3) in detail.

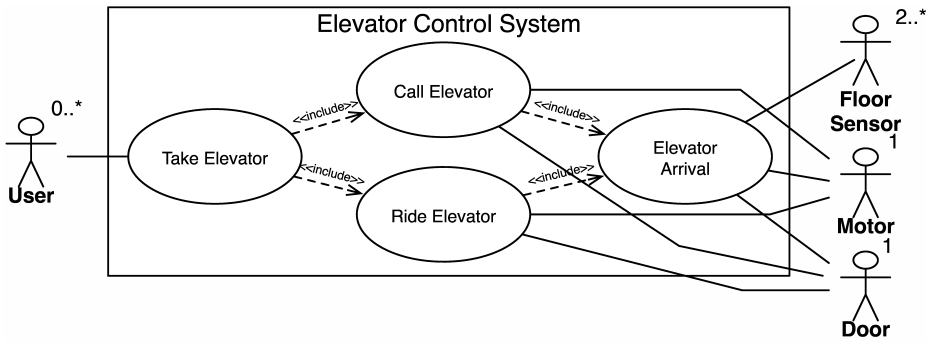

Figure 4.2: Standard Elevator Use Case Diagram

To ride the elevator the *User* enters the cabin, selects a destination floor, waits until the cabin arrives at the destination floor and finally exits the elevator.

CallElevator and *RideElevator* both include the *ElevatorArrival* use case shown in Figure 4.3. It is a subfunction level use case that describes how the system directs the elevator to a specific floor: once the system detects that the elevator is approaching the destination floor, it requests the motor to stop and opens the door.

Use Case: ElevatorArrival
Primary Actor: N/A
Intention: System wants to move the elevator to the *User*'s destination floor.
Level: Subfunction
Main Success Scenario:

1. System asks motor to start moving in the direction of the destination floor;
2. System detects elevator is approaching destination floor;
3. System requests motor to stop;
4. System opens door.

Figure 4.3: *ElevatorArrival* Use Case

The analysis of the basic use case following the approach in [SMKD05] lead to the discovery of some critical exceptions that interrupt the normal elevator arrival processing: *MissedFloor*, *MotorFailure* and *DoorStuckClosed*.

4.3 DA-Charts: Probabilistic Statecharts

In this section, we introduce DA-Charts (short for Dependability Assessment Charts), a probabilistic extension of the Statecharts formalism introduced by David Harel [Har87].

4.3.1 Extending Statecharts with Probabilities

We extend the statecharts formalism with probabilities to enable dependability assessment. While stochastic petri nets [Mar90] is an established formalism with clearly defined semantics, statecharts seem a more natural match for our domain. This, thanks to their modularity, broadcast, and orthogonality features. Statecharts also make it possible to design visually simple and structured models.

Standard statecharts are solely event-driven. State transitions occur if the associated event is

triggered and any specified condition is satisfied. Given the event, a source state has only one possible target state. In the formalism we propose, DA-Charts, when an event is triggered, a state can transition to one of two possible target states: a *success* state and a *failure* state. When an event is triggered, the system moves to a success state with probability p and to a failure state with probability *1-p*. In most real-time systems, the probability of ending up in a success state is closer to 1 and the failure state probability is closer to 0. For example, if a motor in a mechanical system is asked to stop, it might stop with a probability of 0.999 and it might fail to stop with probability 0.001. As in statecharts, the transition may broadcast events. The event that is broadcast can be different depending on whether the transition leads to a success state or a failure state. Hence, the outcome of the event might vary.

DA-Charts Syntax. The statecharts notation is extended to include probabilities. The standard transition is split into two transitions, each annotated with the probability that the event associated with the transition leads to a success state or a failed state. The notation used for this purpose adds an attribute next to the event: *event[condition]* {*probability*} */action*. Absence of the *probability* attribute denotes a probability of 1.

DA-Charts Constraints. Our DA-Charts formalism is constrained by the following:

- Every DA-Chart must contain a *system* component describing the behavior of the software of the system. No probabilities are allowed in the system component, since at the requirements level we assume a fault-free implementation.
- Each secondary actor is modelled by an orthogonal component. Each service that an actor provides can either succeed or fail, which is modelled by two transitions leading to either a *success* or a *failed* state, annotated with the corresponding probabilities.
- To monitor the safety constraints of the system, an additional orthogonal *safety-status* component is created. Whenever the failure of an actor leads to an unsafe condition, a *toUnsafe* event is broadcast to the *safety-status* component. Other quality constraints can be modelled in a similar manner.

4.3.2 DA-Charts Implementation in AToM^3

DA-Charts is a simple extension of the statechart syntax: a simple edge is extended by adding a *probability* attribute which becomes a P-Edge, so the action and the target depend on the outcome of a probabilistic experiment. A traditional edge can be seen as a P-Edge whose probability is 1.

We implement tool support for DA-Charts by extending the meta-model of the DCharts formalism (a variant of Statecharts) described in [Hui04]. This is done in three steps as follows. First, *probability* is added as a float attribute to the *Hyperedge* relationship of the existing DCharts meta-model (an Entity-Relationship diagram). The default value of *probability* is 1. Two constraints are added. One constraint allows users to only set the probability of a transition to a maximum of 1; the other one checks if the total probability of all transitions from the same source node and triggered by the same event is 1. AToM^3 [dLV02a, dLVA04a] allows for the subsequent synthesis of a visual DA-Charts modeling environment from this meta-model. Second, a Probability Analysis (PA) module which can compute probabilities of reaching a target state is implemented. The algorithm is described in the next section. Lastly, a button which invokes the PA module is added to the visual modeling environment.

The semantics of a DA-Chart are described informally as follows. When an event occurs, all P-Edges which are triggered by the event and whose guards hold are taken. The system then leaves the source node(s), chooses one of those P-Edges probabilistically, executes the action of the chosen P-Edge, and enters the target node(s).

4.3.3 Probability Analysis of DA-Charts in AToM^3

Given a source state (consisting of a tuple of source nodes) and a target state, the probability to reach the target from the source is computed by finding all paths that lead from the source to the target state. The probability of each path is calculated as the product of all transition probabilities. The total probability is then computed by adding the probabilities of all paths.

A probabilistic analysis algorithm based on the above observations has been implemented in AToM^3. It reads two arguments, model M containing all elements, such as components, nodes and edges, and a tuple of node names of the target state T. It then produces a float value in the range $[0, 1]$. The algorithm is shown in Algorithm 10.

An analyst wanting to compute, for instance, the reliability of the system has to first press the *PA* button, and then select the target state that symbolizes the successful completion of the goal, after which a pop-up dialog shows the result and all possible paths leading to the target state are highlighted in the model.

Figure 4.4 shows an example DA-Chart model in AToM^3. The model consists of three components: *System*, *D1* and *D2*. The default state is *(s5, s1, s8)* and the only transition which can happen initially is the one from *s1* to *s2*. The probability of reaching *(s3, *)* ("*" means we do not care about what other nodes are when the system ends in the state containing *s3*) from *(s5, s1, s8)* is 99.95% which is the combination of the probabilities along two possible paths: $T_{s1\to s2}$, $T_{s5\to s6}$, and $T_{s2\to s3}$ for path one; $T_{s1\to s2}$, $T_{s5\to s7}$, $T_{s2\to s4}$, $T_{s8\to s9}$, and $T_{s4\to s3}$ for path two. The computation performed can be mathematically defined as follows:

$$P_{total} = P_{s2\to s3} \times P_{s5\to s6} \times P_{s1\to s2} + (P_{s4\to s3} \times P_{s8\to s9}) \times P_{s2\to s4} \times P_{s5\to s7} \times P_{s1\to s2} \quad (4.1)$$

4.3.4 Mapping Exceptional Use Cases to Sequence Diagrams and DA-Charts

We assume that the system software and the communication channels between the system and the actors are reliable. During requirements elicitation, the developer can assume that the system itself, once it has been built, will always behave according to specification - in other words, it will not contain any faults, and will therefore never fail. As the development continues into design and implementation phases, this assumption is most certainly not realistic. Dependability assessment and fault forecasting techniques have to be used to estimate the reliability of the implemented system. If needed, fault tolerance mechanisms have to be built into the system to increase its dependability.

Although the system is assumed to function perfectly, a reliable system cannot assume that it will operate in a fault free environment. Hence, at this point we need to consider the possible failure of (secondary) actors to perform the services requested by the system that affects the dependability of the system.

First, each use case is mapped to one Sequence Diagram and subsequently transformed into a Statechart. Then each Statechart is annotated with probabilities to become one DA-Chart. As mentioned above, the DA-Chart has one orthogonal *system* component that models the behavior of the system, one *safety-status* component that records unsafe states, and one probabilistic orthogonal component for each secondary actor.

Algorithm 10 probAnalysis(M, T)

```
a2t_map = {}
for all node in M do
  for all transition starting from node do
    action = getActionTriggered(transition)
    a2t_map[action] = transition
  end for
end for
p = 0
for all node in T do
  transitions = getIncomeLinks(node)
  for all transition in transitions do
    p += calculateProb(transition, a2t_map)
  end for
end for
return p
## computes accumulated probability starting from target node
calculateProb(transition, a2t_map):
p = 0
(trigger, prob, action) = parseTransition(transition)
prevNode = getIncomeNode(transition)
if isNull(trigger) then
  transitions = getIncomeLinks(prevNode)
  if isEmpty(transitions) then
    p = 1
  else
    for all t in transitions do
      p += prob * calculateProb(t, a2t_map)
    end for
  end if
else
  transition = a2t_map[trigger]
  p += prob * calculateProb(transition, a2t_map)
end if
return p
```

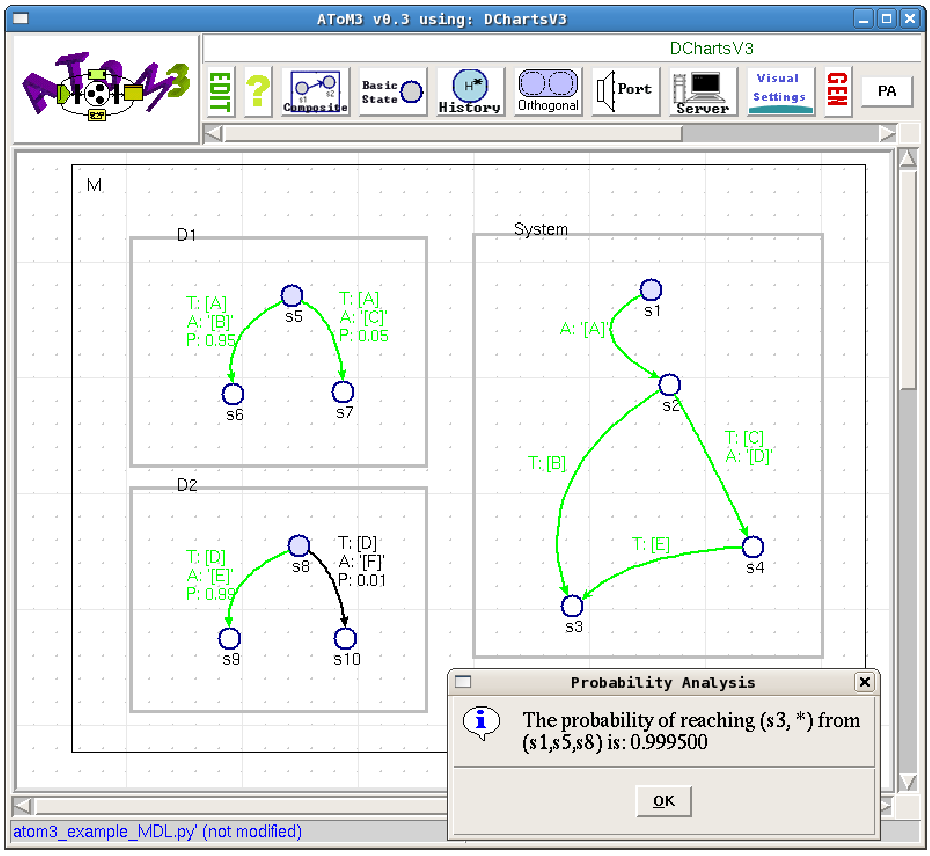

Figure 4.4: Example DA-Chart model in AToM^3

Each step in the use case is first mapped to a message (and finally mapped to a transition) in the system component, as well as a message in the actor involved in the step as follows:

- An appropriately named message is created and transformed to a transition, e.g. *start* or *stop*.
- A step that describes an input sent by an actor A to the system is mapped to:
  - an outgoing message, e.g. *apFlrSnsrDetected*, sending from the component modeling the reliability of A. In the generated Statecharts model, the probability annotation p from the step is added to the success transition, the probability *1-p* is added to the failure transition.
  - an incoming message in the system that moves the system to the next state.
- A step that describes an output sent by the system to an actor A is mapped to:
  - an outgoing message in the system, e.g. *start* or *stop*.
  - an incoming message within the component modeling the behavior of A, that leads to a success state and a failure state. In the generated Statecharts model, probability annotation p from the step is added to the success transition, the probability *1-p* is added to the failure transition.
- A step that describes an output sent by an actor A to an actor B is mapped to:
  - an outgoing message within the component modeling the behavior of A, e.g. *startAck* or *stopAck*.
  - an incoming message within the component modeling the behavior of B, e.g. *startAck* or *stopAck*.
- Each exception associated with the step is mapped to a failure message, e.g. *motorFailure* or *atFlrSnsrFailure*, sending from the corresponding actor to the system.
- If a step is tagged as *Safety-critical*, the actor who sends a failure message may also send a message *toUnsafe* to the *safety-status* component, and the actor who sends a success message may also send a message *toSafe* to the *safety-status* component.

To model the randomness of the system in Sequence Diagrams, we take advantage of one characteristic of Alternative combination fragments. That is, according to UML2.0, if more than one operand of an alternative combination fragment is true, one of them is selected nondeterministically for execution.

4.4 Dependability Analysis Using DA-Charts

4.4.1 Analyze Exceptions in the Elevator Arrival Use Case

We use the Elevator System case study to demonstrate our assessment approach. At this point, the standard use case has been already analyzed for exceptional situations that can arise while servicing a request. As discussed in Section 4.2.1, several failures might occur: the destination floor might not be detected (*MissedFloor*); the motor might fail (*MotorFailure*); or the door might not open at the floor (*DoorStuckClosed*).

Use Case: ElevatorArrival
Intention: System wants to move the elevator to the *User*'s destination floor.
Level: Subfunction
Main Success Scenario:
1. System asks motor to start moving towards the destination floor.
2. System detects elevator is approaching destination floor. Reliability:0.98 *Safety-critical*
3. System requests motor to stop. Reliability:0.99 *Safety-critical*
4. System receives confirmation elevator is stopped at destination floor. Reliability:0.95
5. System requests door to open.
6. System receives confirmation that door is open.

Extensions:
2a. Exception{MissedFloor}
4a. Exception{MotorFailure}
6a. Exception{DoorStuckClosed}

Figure 4.5: Updated *ElevatorArrival* Use Case

To detect whether the elevator is approaching a floor, we need to introduce a sensor, *ApprFloorSensor*. To detect a motor failure, an additional sensor, *AtFloorSensor* is added. It detects when the cabin stopped, and therefore when it is safe to open the doors.

Figure 4.5 shows the updated version of the *ElevatorArrival* use case that includes the added acknowledgment steps and the exception extensions. Some steps are annotated with (made up) probabilities of success: the *ApprFloorSensor* and the *AtFloorSensor* have failure chances of 2% and 5% respectively. The motor has a 1% chance of failure. For space reasons, we assume that the motor always starts and the door always opens. In addition, each step is tagged as *Safety-critical* if the failure of that step threatens the system safety.

4.4.2 Iteration One: Modeling the Basic Elevator Arrival Use Case with Failures and Evaluating Dependability

We first model the initial *ElevatorArrival* use case shown in Section 4.4.1 as a Sequence Diagram without giving probabilities following the process described in Section 4.3.4, as shown in Figure 4.6. Then the model is automatically transformed into a Statechart by means of the Graph Grammar rules we defined in Section 3. And finally, we annotate the transformed model as a DA-Chart with probabilities. The result is shown in Figure 4.7 (we modified the automatically generated names of some states with more meaningful ones). The model consists mainly of four orthogonal components which model the behavior of the system (System), a motor (Motor), and two sensors (*ApprFloorSensor* and *AtFloorSensor*). An additional orthogonal component is used to monitor the safety outcome of the system. Note that the system has no randomness.

To clarify the model, one of the components is briefly explained here. The *Motor* is initially ready (in the *mtr_ready* state). After it is triggered by the *System* (by the *start* event), it acknowledges the *System*'s request (by broadcasting *startAck*) and goes into running mode (by transitioning to the *mtr_started* state). When the motor is asked to stop (by the *stop* event), the *Motor* will either stop itself successfully (going to *mtr_stopped*) and send an acknowledgment (by broadcasting *stopAck*), or fail to stop (going to *mtr_failed* and broadcasting *motorFailure*). The chances of success and failure are 99% and 1% respectively. Before we add these probabilities to the DA-Chart model, we need to model this non-deterministic choice in the Sequence Diagram model. As we mentioned in Section 4.3.4, we model this by means of alternative combination fragments. First, in case of the *Motor*, after it receives the *stop* message from the *System*, we

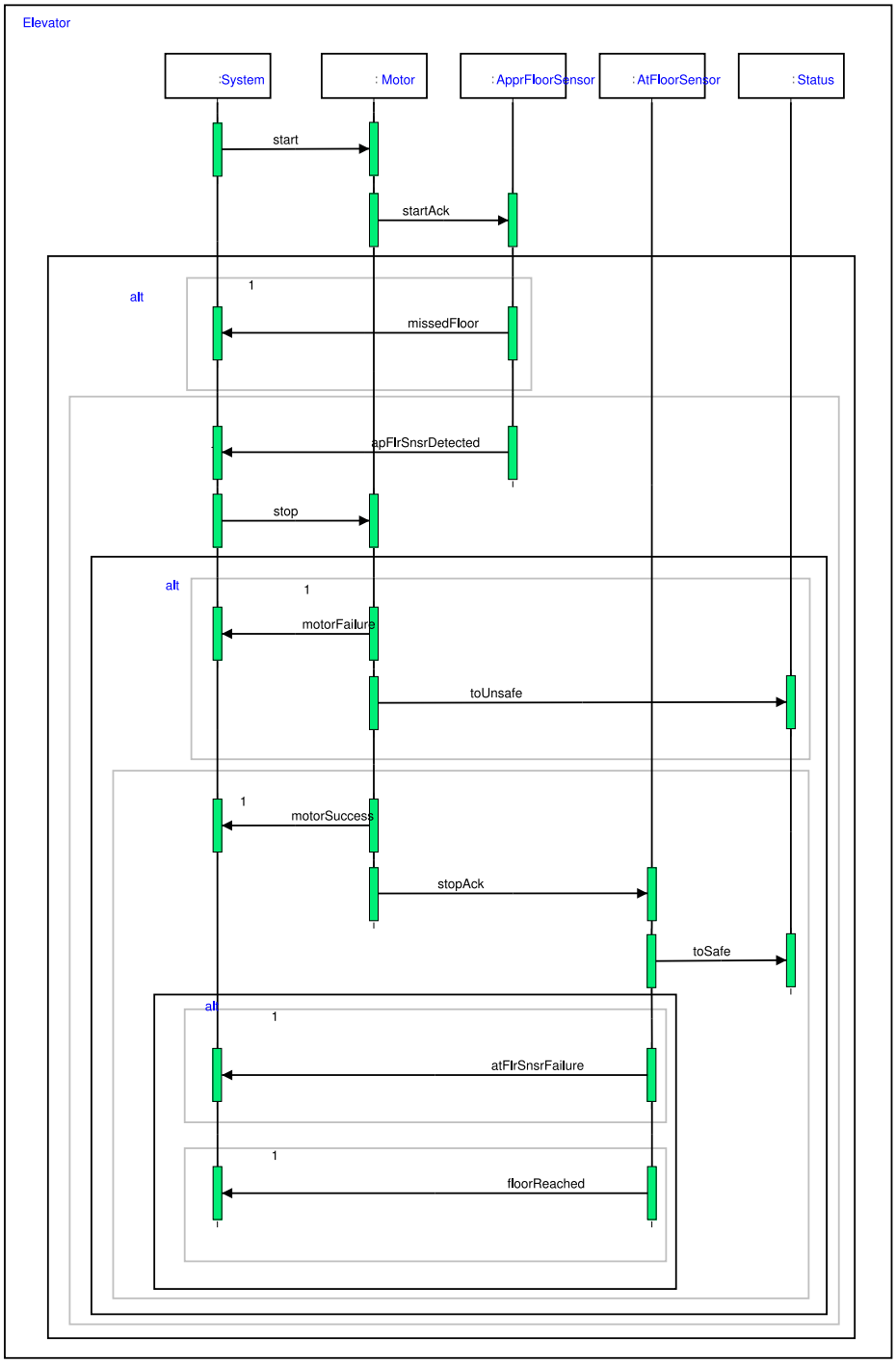

Figure 4.6: Sequence Diagrams Model of the Elevator Arrival Use Case with Failures

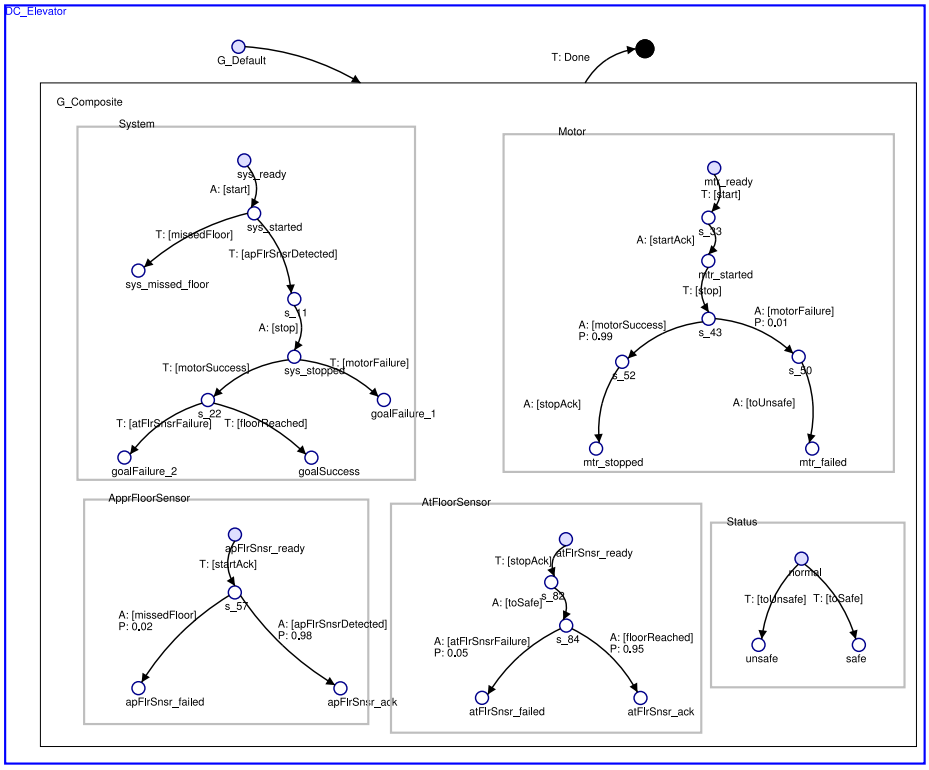

Figure 4.7: DA-Charts Model of the Elevator Arrival Use Case with Failures

use an alternative combined fragment with two interaction operands whose guards are both set to *true* (or 1), to enclose the random branches of the following events from this point. As shown in Figure 4.6, messages starting from *motorFailure* and ending before *motorSuccess* are enclosed in one interaction operand, and following messages are enclosed in another operand of the same alternative combination fragment.

Safety Analysis.

We want to ensure the safety levels maintained by the elevator arrival system. The system is unsafe if the approaching floor sensor fails to detect the destination floor (because then the system never tells the motor to stop), or if the motor fails to stop when told to do so. This is why the failure transition in the *ApprFloorSensor* component, as well as the failure transition in the *Motor* component broadcast a *toUnsafe* event that is recorded in the *Status* component. It is interesting to note that actually achieving the goal of the use case has nothing to do with safety. Our tool then calculates that the probability of reaching the state *safe* from the initial system state (*sys_ready*) is 97.02%, which is the combination of the probabilities along two possible paths.

Reliability Analysis.

Our tool calculates a reliability (probability of reaching the *goalSuccess* state) of 92.169%. Although we assume that the door is 100% reliable, a failure of the *AtFloorSensor* would prevent the system from knowing that the destination floor is reached, and hence the system cannot request the door to open. The person riding the elevator would be stuck inside the cabin, and hence the goal fails.

Requirement Generation from Sequence Diagrams.

The automatic generation of requirements text from the Sequence Diagrams model (shown in Figure 4.6) is shown below.

```
Use Case: Elevator
Actors: System, Motor, ApFlrSnsr, AtFlrSnsr, Status
Scenario:
 1. System sends message 'start' to Motor
 2. Motor sends message 'startAck' to ApFlrSnsr
 3. If 1:
 4.   ApFlrSnsr sends message 'missedFloor' to System
 5. Else if 1:
 6.   ApFlrSnsr sends message 'apFlrSnsrDetected' to System
 7.   System sends message 'stop' to Motor
 8.   If 1:
 9.     Motor sends message 'motorFailure' to System
10.     Motor sends message 'toUnsafe' to Status
11.   Else if 1:
12.     Motor sends message 'motorSuccess' to System
13.     Motor sends message 'stopAck' to AtFlrSnsr
14.     AtFlrSnsr sends message 'toSafe' to Status
15.     If 1:
16.       AtFlrSnsr sends message 'atFlrSnsrFailure' to System
```

Handler Use Case: EmergencyBrake
Handler Class: Safety
Context & Exception: ElevatorArrival{MotorFailure}
Intention: System wants to stop operation of elevator and secure the cabin.
Level: Subfunction
Main Success Scenario:

1. System stops motor.
2. System activates the emergency brakes. Reliability:0.999 *Safety-critical*
3. System turns on the emergency display.

Figure 4.8: *EmergencyBrake* Handler Use Case

```
17.     Else if 1:
18.       AtFlrSnsr sends message 'floorReached' to System
```

4.4.3 Iteration Two: Modeling the Refined Safety-Enhanced Elevator Arrival Use Case and Evaluating Dependability

For a safety-critical system like the elevator control system, a higher level of safety is desirable. Safety can be increased by using more reliable or replicated hardware, but such hardware might not be available or might be too costly. Another possibility is to initiate an action that can prevent catastrophes from happening. To illustrate this approach, we focus on the motor failure problem. To remain in a safe state even if the motor fails, it is necessary to use additional hardware like an emergency brake. This behavior is encapsulated in the *EmergencyBrake* safety handler (shown in Figure 4.8).

The sequence diagram model of the elevator arrival system is updated to reflect the use of emergency brakes (see Figure 4.9), which is then automatically transformed and manually annotated into the DA-Chart model (see Figure 4.10). Another orthogonal component to model the behavior of the emergency brakes is added. The brake used has a 99.9% chance of success.

Safety Analysis.

A probability analysis of the updated model shows a significant improvement in the safety achieved by the system. It is now safe 97.999% (probability of reaching the state *safe_1* and *safe_2*) of the time, which evaluates to an increase of 0.979%. The safety would be even more improved if the *missedFloor* exception were detected and handled.

Reliability Analysis.

The reliability of the system has not changed. The use case could be further refined so that the elevator detects when the *AtFloorSensor* fails, and then the system could redirect the elevator to the nearest floor. Even though the original goal of the user is not satisfied, the system attempts to provide reliable service in a degraded manner.

Requirement Generation from Sequence Diagrams.

The automatic generation of requirements text from the Sequence Diagrams model (shown in Figure 4.9) is shown below.

```
Use Case: Elevator
Actors: System, Motor, ApFlrSnsr, AtFlrSnsr, Status, EmergBrk
```

```
Scenario:
 1. System sends message 'start' to Motor
 2. Motor sends message 'startAck' to ApFlrSnsr
 3. If 1:
 4.   ApFlrSnsr sends message 'missedFloor' to System
 5. Else if 1:
 6.   ApFlrSnsr sends message 'apFlrSnsrDetected' to System
 7.   System sends message 'stop' to Motor
 8.   If 1:
 9.     Motor sends message 'motorFailure' to System
10.     System sends message 'activeEB' to EmergBrk
11.     If 1:
12.       EmergBrk sends message 'toSafe_1' to Status
13.     Else if 1:
14.       EmergBrk sends message 'toUnsafe' to Status
15.   Else if 1:
16.     Motor sends message 'motorSuccess' to System
17.     Motor sends message 'stopAck' to AtFlrSnsr
18.     AtFlrSnsr sends message 'toSafe_2' to Status
19.     If 1:
20.       AtFlrSnsr sends message 'atFlrSnsrFailure' to System
21.     Else if 1:
22.       AtFlrSnsr sends message 'floorReached' to System
```

4.4.4 Discussion

Assessment and refinement is supposed to be an iterative process, and can be continued as long as it is realistic and feasible, until the derived system safety and reliability are met. Supported by our model-driven approach, iterations are correct and not time-consuming, and immediate feedback is given to the analyst of how changes in the use cases affect system dependability.

As we mentioned in Section 4.2, we do not give probabilities in Sequence Diagrams, otherwise there would be direct transformation from a Sequence Diagram model to a DA-Chart model. The main reason we do not add probability to Sequence Diagrams is we have not found a way to formally define the probability and its precise meaning in Sequence Diagram at the moment of writing this thesis. This will become one of subjects of our future work.

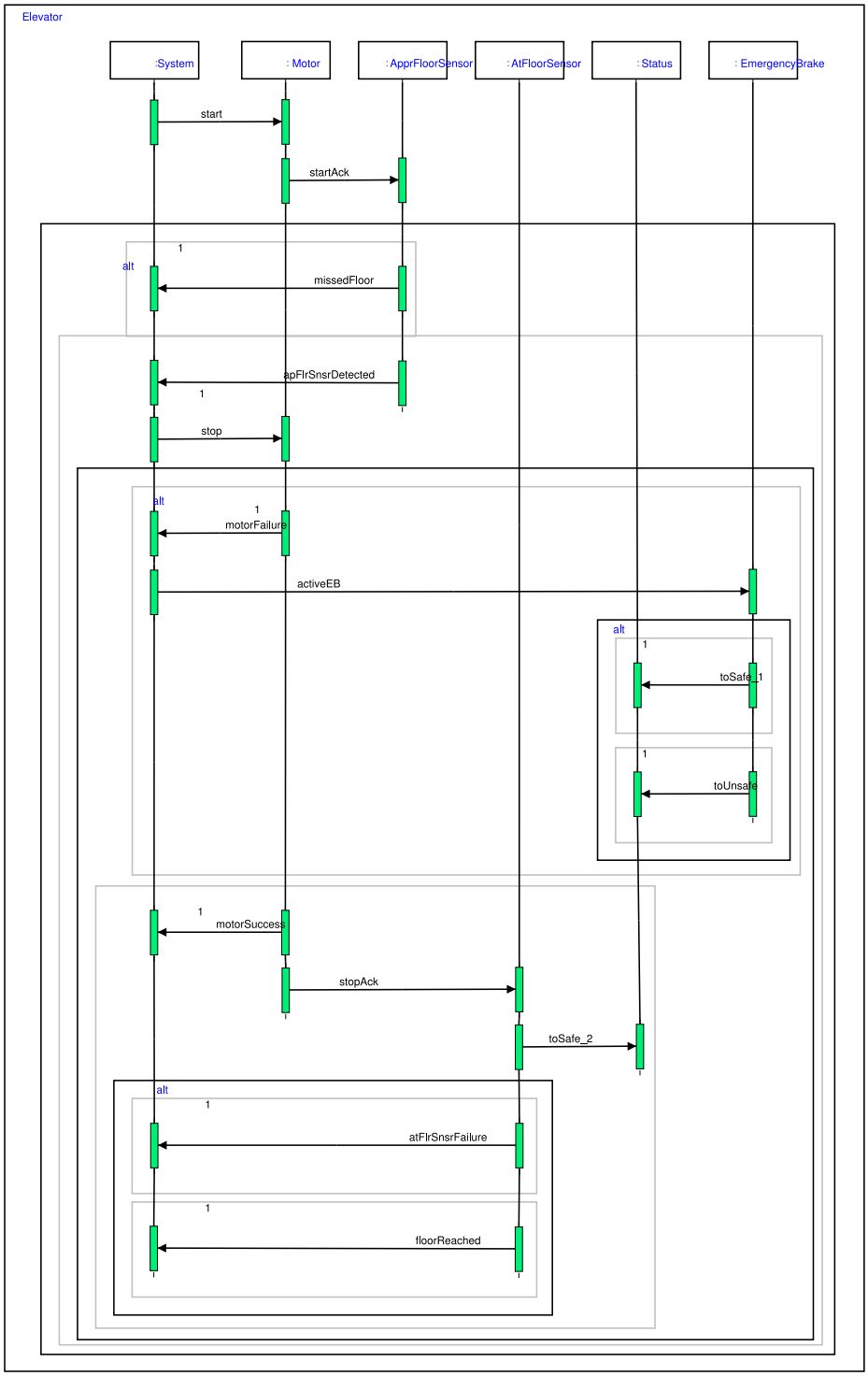

Figure 4.9: Sequence Diagrams Model of the Elevator Arrival System with Failures and Handlers

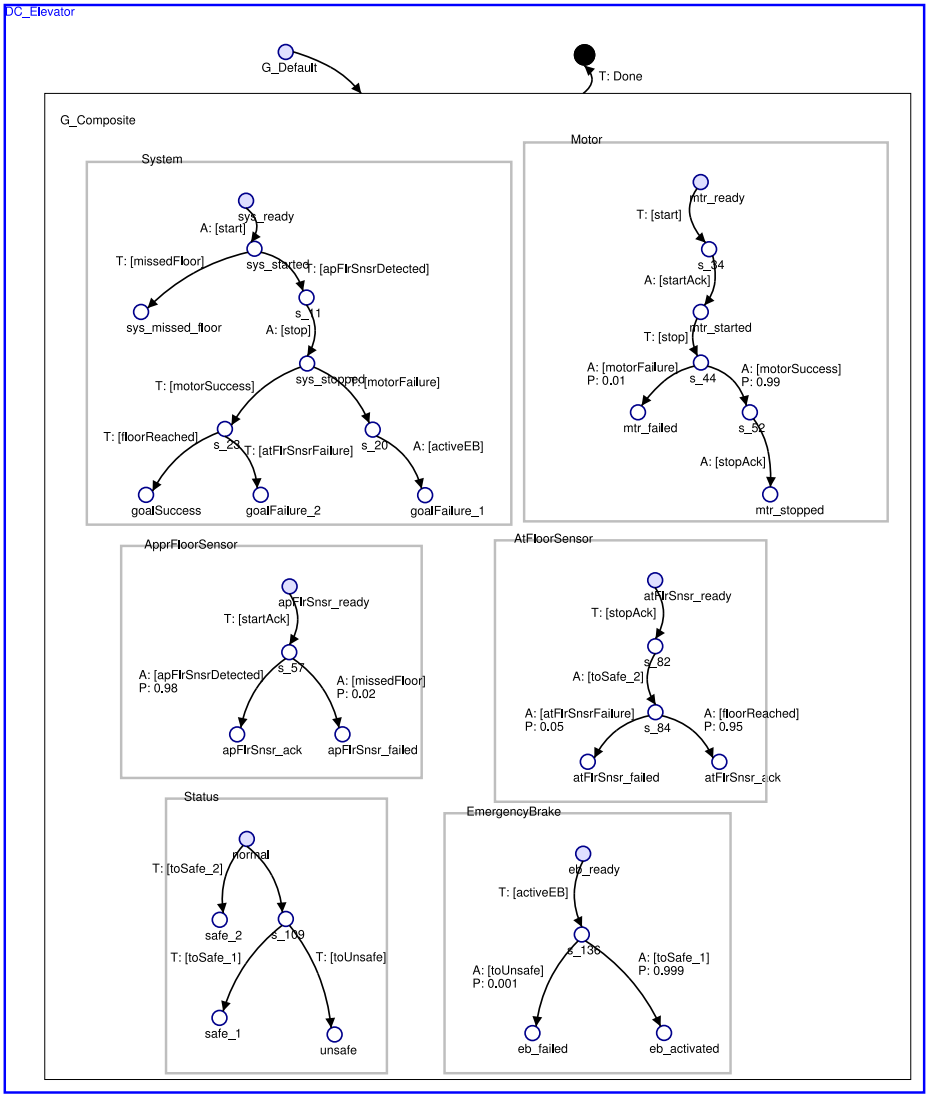

Figure 4.10: DA-Charts Model of the Elevator Arrival System with Failures and Handlers

5
Related Work

Requirements engineering is concerned with the acquisition, analysis, specification, validation, and management of requirements of the software system under construction. Scenarios and use cases are the most important tools used in requirements elicitation. Since scenario-based requirements engineering has been advocated as an effective means of improving the process of requirements engineering, many methodologies and tools [SMMM98, Li00, Dav03, HKP05, WJ06, Whi05] are developed to try to formalize and automate some stages of this process.

Sutcliffe et. al. [SMMM98] 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.

Li [Li00] 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.

Harel et. al. [Dav03, HKP05] 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) [DH01], and finally semi-automatically synthesizing a collection of finite state machines.

Whittle et. al. [WJ06, Whi05] developed a scenario-based language called Use Case Charts (UCCs) to capture scenario-based requirements and presented algorithms for automatic generation of hierarchical state machines from scenario-based models.

Two key activities are apparent in the Harel's and Whittle's approaches: to find a proper scenario-based language, e.g., LSCs or UCCs, to capture and formalize requirements; to transform scenarios into an executable form, i.e., state machines, which can be easily used for requirements simulation, validation and analysis. Both of these are clearly model-driven approaches.

6

Conclusions and Future Work

In this thesis we proposed a model-driven approach to scenario-based requirements engineering. The approach, which is an application of Computer Automated Multi-Paradigm Modeling (CAMPaM), aims to improve the software process. The model-driven approach starts with modeling requirements of a system in scenario models in particular, Sequence Diagrams, and the subsequent automatic transformation to state-based behavior models in particular, Statecharts. Then, either code can be synthesized or models can be further transformed into models with additional information such as explicit timing information or interactions between components. These models, together with the inputs (e.g., queries, performance metrics, test cases, etc.) generated directly from the scenario models, can be used for a variety of purposes, such as verification, analysis, simulation, animation and so on.

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

An application of the model-driven approach to the assessment of use cases for dependable systems was shown. In particular, the case study shows how the model-driven approach helps developers to model and analyze the dependability of use cases and to discover more reliable and safe ways of designing the interactions with the system and the environment.

In the future, the other parts of the model-driven approach will be studied and implemented. For example, the transformation from HSMs to kiltera and the subsequent transformation from kiltera to other formalisms (e.g., TPN, CSP, DEVS); the generation of queries, performance metrics and test cases from the scenario models. It would be useful to build two-way transformations between formalisms. For example, the transformation from Statechart models to Sequence Diagram models. It would be preferable to adopt the Use Case Charts formalism for scenario modeling. Use Case Charts provide a higher level of abstraction for capturing the functionality of a system than using Sequence Diagrams alone. It adds two more abstraction levels above Sequence Diagram: Scenario Chart (used to group and relate a set of Sequence Diagram models) and Use Case Chart (used to group and relate a set of Scenario Chart models).

Bibliography

[ALR01] A. Avizienis, J. Laprie, and B. Randell. Fundamental concepts of dependability, 2001.

[BD03] Bernd Bruegge and Allen H. Dutoit. *Object-Oriented Software Engineering: Using UML, Patterns and Java, Second Edition*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2003.

[BV03] Spencer Borland and Hans L. Vangheluwe. Transforming Statecharts to DEVS. In A. Bruzzone and Mhamed Itmi, editors, *Summer Computer Simulation Conference. Student Workshop*, pages S154 – S159. Society for Computer Simulation International (SCS), July 2003. Montréal, Canada.

[CLOP02] Gennaro Costagliola, Andrea De Lucia, Sergio Orefice, and Giuseppe Polese. A classification framework to support the design of visual languages. *J. Vis. Lang. Comput.*, 13(6):573–600, 2002.

[Dav03] David Harel and Rami Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, 2003.

[Den06] Denis Dubé. Graph Layout for Domain-Specific Modeling. Master's thesis, McGill University, 2006.

[DH01] Werner Damm and David Harel. Lscs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[dLGV04] Juan de Lara, Esther Guerra, and Hans L. Vangheluwe. Meta-Modelling, Graph Transformation and Model Checking for the Analysis of Hybrid Systems. In J.L. Pfaltz, M. Nagl, and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE 2003)*, Lecture Notes in Computer Science 3062, pages 292 – 298. Springer-Verlag, 2004. Charlottesville, Virginia, USA.

[dLV02a] Juan de Lara and Hans Vangheluwe. AToM^3: A tool for multi-formalism and meta-modelling. In *FASE*, pages 174–188, 2002.

[dLV02b] Juan de Lara and Hans L. Vangheluwe. Computer aided multi-paradigm modelling to process petri-nets and statecharts. In *International Conference on Graph Transformations (ICGT)*, volume 2505 of *Lecture Notes in Computer Science*, pages 239–253. Springer-Verlag, October 2002. Barcelona, Spain.

[dLV02c] Juan de Lara and Hans L. Vangheluwe. Using AToM^3 as a Meta-CASE tool. In *4th International Conference on Enterprise Information Systems (ICEIS)*, pages 642 – 649, April 2002. Ciudad Real, Spain.

[dLV02d] Juan de Lara and Hans L. Vangheluwe. Using meta-modelling and graph grammars to process GPSS models. In Hermann Meuth, editor, 16th *European Simulation Multi-conference (ESM)*, pages 100–107. Society for Computer Simulation International (SCS), June 2002. Darmstadt, Germany.

[dLV04] Juan de Lara and Hans L. Vangheluwe. Defining visual notations and their manipulation through meta-modelling and graph transformation. *Journal of Visual Languages and Computing*, 15(3 - 4):309–330, June - August 2004. Special Issue on Domain-Specific Modeling with Visual Languages.

[dLV05] Juan de Lara and Hans L. Vangheluwe. *Model-Based Development: Meta-Modelling, Transformation and Verification*, page 17 pp. The Idea Group Inc., October 2005.

[dLVA04a] Juan de Lara, Hans Vangheluwe, and Manuel Alfonseca. Meta-modelling and graph grammars for multi-paradigm modelling in AToM^3. *Software and System Modeling*, 3(3):194–209, 2004.

[dLVA04b] Juan de Lara, Hans L. Vangheluwe, and Manuel Alfonseca. Meta-modelling and graph grammars for multi-paradigm modelling in AToM^3. *Software and Systems Modeling (SoSyM)*, 3(3):194–209, August 2004.

[dLVAM03] Juan de Lara, Hans L. Vangheluwe, and Manuel Alfonseca Moreno. Computer Aided Multi-Paradigm Modelling of Hybrid Systems with AToM^3. In A. Bruzzone and Mhamed Itmi, editors, *Summer Computer Simulation Conference*, pages 83 – 88. Society for Computer Simulation International (SCS), July 2003. Montréal, Canada.

[Ern07] Ernesto Posse and Hans Vangheluwe. kiltera: a simulation language for timed, dynamic structure systems. In *40th Annual Simulation Symposium*, 2007.

[Fen03] Thomas Huining Feng. An extended semantics for a Statechart Virtual Machine. In A. Bruzzone and Mhamed Itmi, editors, *Summer Computer Simulation Conference. Student Workshop*, pages S147 – S166. Society for Computer Simulation International (SCS), July 2003. Montréal, Canada.

[GM02] Jean-Claude Geffroy and Gilles Motet. *Design of Dependable Computing Systems.* Kluwer Academic Publishers, 2002.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[HKP05] David Harel, Hillel Kugler, and Amir Pnueli. Synthesis revisited: Generating statechart models from scenario-based requirements. In *Formal Methods in Thanks a lot! Software and Systems Modeling*, pages 309–324, 2005.

[HR00] D. Harel and B. Rumpe. Modeling languages: Syntax, semantics and all that stuff, part i: The basic stuff. Technical report, Jerusalem, Israel, 2000.

[Hui04] Huining Feng. DCHARTS, A FORMALISM FOR MODELING AND SIMULATION BASED DESIGN OF REACTIVE SOFTWARE SYSTEMS. Master's thesis, McGill University, 2004.

[JdLMny] Hans Vangheluwe Juan de Lara and Pieter J. Mosterman. Modelling and analysis of traffic networks based on graph transformation. *Formal Methods for Automation and Safety in Railway and Automotive Systems*, page 11, December 2004. Braunschweig, Germany.

[LAK92] J.C. C. Laprie, A. Avizienis, and H. Kopetz, editors. *Dependability: Basic Concepts and Terminology*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1992.

[Lar02] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. 2nd edition, 2002.

[Li00] Liwu Li. Translating use cases to sequence diagrams. In *ASE*, pages 293–296, 2000.

[LJVdLM04] Simon Lacoste-Julien, Hans L. Vangheluwe, Juan de Lara, and Pieter J. Mosterman. Meta-modelling hybrid formalisms. In Pieter J. Mosterman and Jin-Shyan Lee, editors, *IEEE International Symposium on Computer-Aided Control System Design*, pages 65 – 70. IEEE Computer Society Press, September 2004. Taipei, Taiwan. **Invited** paper.

[Mar90] Marco Ajmone Marsan. Stochastic petri nets: an elementary introduction. In *Advances in Petri Nets 1989, covers the 9th European Workshop on Applications and Theory in Petri Nets-selected papers*, pages 1–29, London, UK, 1990. Springer-Verlag.

[MLS05] Unified Modeling Language 2.0 Specification. http://www.omg.org, 2005.

[MSKV06] Sadaf Mustafiz, Ximeng Sun, Jörg Kienzle, and Hans Vangheluwe. Model-driven assessment of use cases for dependable systems. In *MoDELS*, pages 558–573, 2006.

[NE00] Bashar Nuseibeh and Steve Easterbrook. Requirements engineering: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 35–46, New York, NY, USA, 2000. ACM Press.

[PB03] Ernesto Posse and Jean-Sébastien Bolduc. Generation of DEVS modelling and simulation environments. In A. Bruzzone and Mhamed Itmi, editors, *Summer Computer Simulation Conference. Student Workshop*, pages S139 – S146. Society for Computer Simulation International (SCS), July 2003. Montréal, Canada.

[PdLV02] Ernesto Posse, Juan de Lara, and Hans L. Vangheluwe. Processing causal block diagrams with graph-grammars in AToM^3. In *European Joint Conference on Theory and Practice of Software (ETAPS), Workshop on Applied Graph Transformation (AGT)*, pages 23 – 34, April 2002. Grenoble, France.

[RC95] Mary Beth Rosson and John M. Carroll. Narrowing the specification-implementation gap in scenario-based design. pages 247–278, 1995.

[Rha] Rhapsody. I-Logix, Inc., http://www.ilogix.com/products/.

[SMKD05] Aaron Shui, Sadaf Mustafiz, Jörg Kienzle, and Christophe Dony. Exceptional use cases. In *MoDELS*, pages 568–583, 2005.

[SMMM98] Alistair G. Sutcliffe, Neil A. Maiden, Shailey Minocha, and Darrel Manuel. Supporting scenario-based requirements engineering. *Software Engineering*, 24(12):1072–1088, 1998.

[Sut03] Alistair Sutcliffe. Scenario-based requirements engineering. *11th IEEE International Requirements Engineering Conference*, 00:320, 2003.

[VdL02] Hans L. Vangheluwe and Juan de Lara. Meta-models are models too. In E. Yücesan, C.-H. Chen, J.L. Snowdon, and J.M. Charnes, editors, *Winter Simulation Conference*, pages 597 – 605. IEEE Computer Society Press, December 2002. San Diego, CA. **Invited** paper.

[VdL03] Hans Vangheluwe and Juan de Lara. Foundations of multi-paradigm modeling and simulation: computer automated multi-paradigm modelling: meta-modelling and graph transformation. In *Winter Simulation Conference*, pages 595–603, 2003.

[VdL04] Hans Vangheluwe and Juan de Lara. Computer automated multi-paradigm modelling for analysis and design of traffic networks. In *Winter Simulation Conference*, pages 249–258, 2004.

[Whi05] Jon Whittle. Specifying precise use cases with use case charts. In *MoDELS Satellite Events*, pages 290–301, 2005.

[WJ06] Jon Whittle and Praveen K. Jayaraman. Generating hierarchical state machines from use case charts. *Proceedings of the 14th IEEE International Requirements Engineering Conference (RE'06)*, 0:16–25, 2006.

A

Transformation Trace

The trace of the model transformation applied to the example of Figure 3.2 described in Section 3 is listed as follows:

```
Step   1: initInteraction_GG_rule 0
Step   2: importInteraction_GG_rule 1
Step   3: connectHeadAFFirst_GG_rule 2
Step   4: connectHeadAFFirst_GG_rule 2
Step   5: connectHeadAFSecond_GG_rule 3
Step   6: connectHeadAFSecond_GG_rule 3
Step   7: connectTailAFSecond_GG_rule 5
Step   8: connectTailAFSecond_GG_rule 5
Step   9: createWrapperCF_GG_rule 7
Step  10: connectCFWithWrapperCF_GG_rule 8
Step  11: cleanImportedLifeline_GG_rule 9
Step  12: cleanImportedLifeline_GG_rule 9
Step  13: cleanImportedInteraction_GG_rule 10
Step  14: initInteraction_GG_rule 0
Step  15: interaction_GG_rule 11

Step  16: lifeline_GG_rule 12
Step  17: msgOut_GG_rule 18
Step  18: combinedFragment_GG_rule 13
Step  19: interactionOperand_GG_rule 14
Step  20: setScope_GG_rule 19
Step  21: combinedFragment_GG_rule 13
Step  22: interactionOperand_GG_rule 14
Step  23: interactionOperand_GG_rule 14
Step  24: msgIn_GG_rule 18
Step  25: msgOut_GG_rule 18
Step  26: msgIn_GG_rule 18
Step  27: connetWithFinal_GG_rule 20
Step  28: connetWithFinal_GG_rule 20
Step  29: connetWithFinal_GG_rule 20
Step  30: connetWithFinal_GG_rule 20
Step  31: cleanLifeline_GG_rule 21

Step  32: lifeline_GG_rule 12
```

```
Step  33: msgIn_GG_rule 18
Step  34: combinedFragment_GG_rule 13
Step  35: interactionOperand_GG_rule 14
Step  36: setScope_GG_rule 19
Step  37: combinedFragment_GG_rule 13
Step  38: interactionOperand_GG_rule 14
Step  39: interactionOperand_GG_rule 14
Step  40: msgOut_GG_rule 18
Step  41: msgIn_GG_rule 18
Step  42: msgOut_GG_rule 18
Step  43: msgOut_GG_rule 18
Step  44: connetWithFinal_GG_rule 20
Step  45: connetWithFinal_GG_rule 20
Step  46: connetWithFinal_GG_rule 20
Step  47: connetWithFinal_GG_rule 20
Step  48: cleanLifeline_GG_rule 21

Step  49: lifeline_GG_rule 12
Step  50: msgIn_GG_rule 18
Step  51: setScope_GG_rule 19
Step  52: connetWithFinal_GG_rule 20
Step  53: cleanLifeline_GG_rule 21
Step  54: cleanActionFragment_GG_rule 22
Step  55: cleanActionFragment_GG_rule 22
Step  56: cleanActionFragment_GG_rule 22
Step  57: cleanActionFragment_GG_rule 22
Step  58: cleanActionFragment_GG_rule 22
Step  59: cleanActionFragment_GG_rule 22
Step  60: cleanActionFragment_GG_rule 22
Step  61: cleanActionFragment_GG_rule 22
Step  62: cleanActionFragment_GG_rule 22
Step  63: cleanActionFragment_GG_rule 22
Step  64: cleanInteraction_GG_rule 23
Step  65: cleanCombinedFragment_GG_rule 24
Step  66: cleanCombinedFragment_GG_rule 24
Step  67: cleanInteractionOperand_GG_rule 25
Step  68: cleanInteractionOperand_GG_rule 25
Step  69: cleanInteractionOperand_GG_rule 25

Step  70: orthogonalOptFirst_GG_rule 26
Step  71: orthogonalOptFirst_GG_rule 26
Step  72: orthogonalOptFirst_GG_rule 26
Step  73: orthogonalOptFirst_GG_rule 26
Step  74: orthogonalOptFirst_GG_rule 26
Step  75: orthogonalOptFirst_GG_rule 26
Step  76: orthogonalOptFirst_GG_rule 26
Step  77: orthogonalOptFirst_GG_rule 26
```

```
Step  78: orthogonalOptFirst_GG_rule 26
Step  79: orthogonalOptFirst_GG_rule 26
Step  80: orthogonalOptFirst_GG_rule 26
Step  81: orthogonalOptFirst_GG_rule 26
Step  82: orthogonalOptFirst_GG_rule 26
Step  83: orthogonalOptFirst_GG_rule 26
Step  84: orthogonalOptFirst_GG_rule 26
Step  85: orthogonalOptFirst_GG_rule 26
Step  86: orthogonalOptFirst_GG_rule 26
Step  87: orthogonalOptFirst_GG_rule 26
Step  88: orthogonalOptSecond_GG_rule 27
Step  89: orthogonalOptSecond_GG_rule 27
Step  90: orthogonalOptThird_GG_rule 28
Step  91: orthogonalOptThird_GG_rule 28
Step  92: orthogonalOptThird_GG_rule 28
Step  93: orthogonalOptThird_GG_rule 28
Step  94: orthogonalOptThird_GG_rule 28
Step  95: orthogonalOptThird_GG_rule 28
Step  96: compositeOptFirst_GG_rule 29
Step  97: compositeOptFirst_GG_rule 29
Step  98: compositeOptSecond_GG_rule 30
Step  99: compositeOptSecond_GG_rule 30
Step 100: compositeOptThird_GG_rule 31
Step 101: compositeOptFirst_GG_rule 29
Step 102: compositeOptFirst_GG_rule 29
Step 103: compositeOptThird_GG_rule 31
Step 104: compositeOptSecond_GG_rule 30
Step 105: compositeOptSecond_GG_rule 30
Step 106: compositeOptThird_GG_rule 31
Step 107: compositeOptThird_GG_rule 31
Step 108: compositeOptThird_GG_rule 31
Step 109: compositeOptFirst_GG_rule 29
Step 110: compositeOptFirst_GG_rule 29
Step 111: compositeOptThird_GG_rule 31
Step 112: compositeOptSecond_GG_rule 30
Step 113: compositeOptSecond_GG_rule 30
Step 114: compositeOptThird_GG_rule 31
Step 115: compositeOptThird_GG_rule 31
Step 116: compositeOptFour_GG_rule 32
Step 117: compositeOptFour_GG_rule 32
Step 118: compositeOptFour_GG_rule 32
Step 119: compositeOptFour_GG_rule 32
Step 120: compositeOptFour_GG_rule 32
Step 121: compositeOptFour_GG_rule 32
Step 122: compositeOptFour_GG_rule 32
Step 123: compositeOptFour_GG_rule 32
Step 124: setOptScope_GG_rule 33
```

```
Step 125: compositeOptPrime_GG_rule 34
Step 126: compositeOptSecondPrime_GG_rule 35
Step 127: compositeOptSecondPrime_GG_rule 35
Step 128: compositeOptSecondPrime_GG_rule 35
Step 129: compositeOptSecondPrime_GG_rule 35
Step 130: compositeOptSecondPrime_GG_rule 35
Step 131: compositeOptSecondPrime_GG_rule 35
Step 132: compositeOptSecondPrime_GG_rule 35
Step 133: compositeOptSecondPrime_GG_rule 35
Step 134: compositeOptSecondPrime_GG_rule 35
Step 135: compositeOptSecondPrime_GG_rule 35
Step 136: compositeOptSecondPrime_GG_rule 35
Step 137: compositeOptThirdPrime_GG_rule 36
Step 138: compositeOptFourPrime_GG_rule 37
Step 139: compositeOptPrime_GG_rule 34
Step 140: compositeOptPrime_GG_rule 35
Step 141: compositeOptSecondPrime_GG_rule 35
Step 142: compositeOptSecondPrime_GG_rule 35
Step 143: compositeOptSecondPrime_GG_rule 35
Step 144: compositeOptSecondPrime_GG_rule 35
Step 145: compositeOptSecondPrime_GG_rule 35
Step 146: compositeOptSecondPrime_GG_rule 35
Step 147: compositeOptSecondPrime_GG_rule 35
Step 148: compositeOptSecondPrime_GG_rule 35
Step 149: compositeOptSecondPrime_GG_rule 35
Step 150: compositeOptSecondPrime_GG_rule 35
Step 151: compositeOptThirdPrime_GG_rule 36
Step 152: compositeOptFourPrime_GG_rule 37
Step 153: cleanStateFirstPrime_GG_rule 38
Step 154: cleanStateSecond_GG_rule 38
Step 155: cleanStateFirstPrime_GG_rule 38
Step 156: cleanStateFirst_GG_rule 38
Step 157: cleanStateFirstPrime_GG_rule 38
Step 158: cleanStateFirst_GG_rule 38
Step 159: cleanStateSecond_GG_rule 38
Step 160: cleanStateFirst_GG_rule 38
Step 161: cleanStateFirst_GG_rule 38
Step 162: cleanStateSecond_GG_rule 38
Step 163: cleanStateFirstPrime_GG_rule 38
Step 164: cleanStateFirst_GG_rule 38
Step 165: cleanStateSecond_GG_rule 38
Step 166: cleanStateFirst_GG_rule 38
Step 167: cleanStateFirstPrime_GG_rule 38
Step 168: cleanStateSecond_GG_rule 38
Step 169: cleanStateSecond_GG_rule 38
Step 170: cleanStateFirst_GG_rule 38
Step 171: cleanStateFirstPrime_GG_rule 38
```

```
Step 172: cleanStateFirst_GG_rule 38
Step 173: cleanStateFirstPrime_GG_rule 38
Step 174: cleanStateFirstPrime_GG_rule 38
Step 175: cleanStateFirstPrime_GG_rule 38
Step 176: cleanStateFirstPrime_GG_rule 38
Step 177: cleanStateFirstPrime_GG_rule 38
Step 178: finalstateOptFirst_GG_rule 39
Step 179: finalstateOptFirst_GG_rule 39
Step 180: finalstateOptFirst_GG_rule 39
Step 181: finalstateOptSecond_GG_rule 40
Step 182: finalstateOptSecond_GG_rule 40
Step 183: finalstateOptSecond_GG_rule 40
```