Relationship-Based Change Propagation: A Case Study

by

Winnie Lai

A thesis submitted in conformity with the requirements
for the degree of Master of Computer Science

Department of Computer Science
University of Toronto

Relationship-Based Change Propagation: A Case Study

Winnie Lai

Master of Computer Science

Department of Computer Science
University of Toronto

2009

Abstract

Software development is an evolutionary process. Requirements of a system are often incomplete or inconsistent, and hence need to be extended or modified over time. Customers may demand new services or goals that often lead to changes in the design and implementation of the system. These changes are typically very expensive. Even if only local modifications are needed, manually applying them is time-consuming and error-prone. Thus, it is essential to assist users in propagating changes across requirements, design, and implementation artifacts.

In this thesis, we take a model-based approach and provide an automated algorithm for propagating changes between requirements and design models. The key feature of our work is explicating relationships between models at the requirements and design levels. We formalize the relationships and provide conditions for checking validity of these relationships both syntactically and semantically. We show how our algorithm utilizes the relationships between models at different levels to localize the regions that should be modified. We use the IBM Trade6 case study to demonstrate our approach.

Acknowledgments

I am grateful to Professor Marsha Chechik for granting me an opportunity in working with her on this interesting research project, and her enormous effort in supervising me during the course of research. I would like to thank Shiva Nejati as well for her extra effort in helping me on the research for various areas. I also thank Rick Salay for helping me to understand the MMTF, and Jordi Cabot for providing different perspectives of initial research directions. After all, thanks to every member in the Merge Group.

Table of Contents

List of Figures

1 Introduction

Software development is an evolutionary process. Requirements of a system are often incomplete or inconsistent, and hence need to be extended or modified over time. Customers may demand new services or goals. These often lead to major or minor design and implementation changes. Sometimes these changes trigger a complete redesign or reconfiguration of the underlying system, such as changes in non-functional requirements or system architecture but sometimes the changes have a local effect, requiring developers to modify only a small part of a system. In the latter case, it is essential for developers to separate those parts of the system that are intact, and hence can be reused, from those places that must be modified in response to the change.

The process of modifying software to meet its changing requirements is challenging and has been extensively studied before under terms *software adaptation* [2][3], *software evolution* [4], and *change impact analysis* [5][6]. Software adaptation often refers to designing a system such that it can operate correctly in a changing environment, i.e., facilitating “online” change. In contrast, we study changes that are done “offline”; we assume that changes are made, the system is recompiled and then put back into operation. Typically, such process is referred to as change impact analysis or software evolution.

We take a model-based approach and provide an automated technique for propagating changes between requirements and design models. We start with a collection of models that describe a system at different levels of abstraction and/or from different perspectives. Our goal is to provide a technique for propagating changes across these models. The key feature of our work is to explicate relationships between these models, and then utilize these relationships to propagate changes automatically, if possible, and to localize the regions in other models that should be modified by hand.

The earlier work of members of the Toronto Merge Group[1] has studied relationships between homogeneous models, i.e., models defined in the same notation. In particular, they have characterized syntactic and semantic relationships between structural models, such as class diagrams and ER diagrams [7], and behavioural models, such as state machines [8]. Further, they have developed semi-automated algorithms for computing such relationships [8]. Here, we build on their earlier work to describe relationships between a set of heterogeneous models, i.e., models described in different notations. The syntax and semantics of such relationships are typically specified through mappings between different model types. A relationship between a pair of heterogeneous models is valid if it conforms to the mappings defined between their respective meta-models.

1.1 Related Work

Specifying relationships between a set of heterogeneous models has been studied previously: [9] proposes an approach for checking the logical consistency of a set of related requirements. The consistency rules are described using first-order logic and are checked using a classical theorem prover. [10] develops an end-to-end framework, called xlinkit, for consistency checking of distributed XML documents. The framework includes a document management mechanism, a language based on first-order logic for expressing consistency rules, and a conformance checking engine for verifying documents against these rules and generating diagnostics. While these techniques can efficiently describe relationships across a set of heterogeneous models and can verify consistency of the models and their relationships, they do not provide support for change propagation or model repair in case an inconsistency arises.

---

[1] Merge Group at Software Engineering Group, Department of Computer Science of University of Toronto.

Our work is most closely related to the efforts on impact analysis [6] and change propagation in the context of software engineering models [4]. [6] uses consistency rules to determine, as the change is made, which of the instances need to be reevaluated. [4] explicitly enumerates the types of changes that can be made on a particular type of models and gives recipes of how to propagate each kind of change among a related collection of models. The work is limited to Sequence Diagrams and Class Diagrams. We are not aware of work on automatic repair: while this approach is used in the database research, it does not seem to be applied yet to general software engineering models. Thus, we produce regions with unknowns rather than automatically generating the changed models.

1.2 Contribution

In this thesis, we formalize relationships between heterogeneous models, specifically, activity and sequence diagrams. While these models intrinsically describe a software system from different perspectives, we use them to represent different abstraction levels of the system. This is different than [4] that uses sequence diagrams and class diagrams to represent the same level of abstraction. We give a concept of a *region* to encapsulate a message or a sequence of messages in sequence diagram. This essentially hides the details of a refined model when we focus on relating its elements to the elements of an abstract model. It is obvious that such detailed information is needed when one relates regions to the elements of a more refined model. A region is different from `InteractionFragment` (an element in sequence diagram meta-model) as region gives the flexibility of binding messages – there can be multiple regions in one interaction fragment, though it may be less likely that a region spans across multiple fragments.

In addition, we define rules to validate the relationships between the models. Utilizing these relationships, we provide our algorithm to automatically propagate changes across models. Further, we illustrate how the algorithm works using a fragment of IBM’s Trade6 system, and discuss how it can be implemented in existing model-based tools.

1.3 Organization of This Thesis

The rest of this thesis is organized as follows. In Chapter 2, we describe our model-based change propagation technique by demonstrating it on a case study: an IBM WebSphere Performance Benchmark Sample called Trade6 [1]. Specifically, we show how relationships between heterogeneous models can be defined, and illustrate how these relationships can be used to propagate changes. We then formalize the relationships between heterogeneous models in Chapter 3. We provide our change propagation algorithm, and show how it can help us identify and localize the effects of change across a set of inter-related models in Chapter 4. We briefly discuss how the algorithm can be implemented with existing modeling tools in Chapter 5. Finally, we conclude the thesis with a summary, limitations and a discussion of future research directions in Chapter 6.

2 Use Case and Motivation

In this chapter, we motivate our work using an example of an online brokerage application. We begin by introducing the example and providing a set of inter-related UML diagrams for the example in Section 2.1. We define the relationships between different diagrams in Section 2.2. Finally, we show how these relationships can be utilized for change propagation in Section 2.3.

2.1 Motivating Example

We use an online brokerage application in our study of propagation of model changes. For that, we need two levels of model abstractions to describe requirements and designs respectively. We have found a specification of an online brokerage system [11] that covers fundamental use cases, which include account profile management, placing a trade order, and retrieval of stock information from stock exchange. The specification captures documentation created at different stages of software development: stake holder's requests, use case requirements, and designs of the system. In particular, there is an activity diagram describing the requirements for a use case of placing an order, which describes the flow for both buy and sell orders. For the sake of our example, we simplify the activity diagram and use it as the requirements model for the use case of buy order.

Further, we use Trade6 [1] as our additional sources of models. Trade6 is an online brokerage application designed for benchmarking web service performance. Using java and IBM WebSphere packages, it implements standard use cases that include getting account profile, getting stock quote, buy order, and sell order. Since Trade6 does not provide explicit requirements and design documentation, we reverse engineer a sequence diagram of buy order from code. We use the sequence diagram as the design model for the use case of buy order.

Figure 1 shows the requirements model of the use case of buy order. In this use case, the user has been logged into the system. The user first enters the stock name and the number of

stocks to buy, the order is then placed on the queue for process. Finally, the system notifies the user when the order is finished.

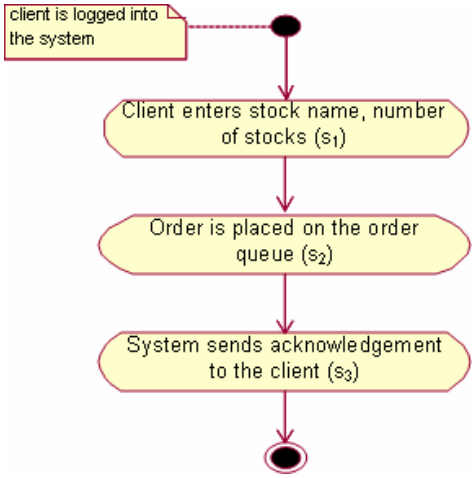

Figure 1. Activity diagram of buy order.

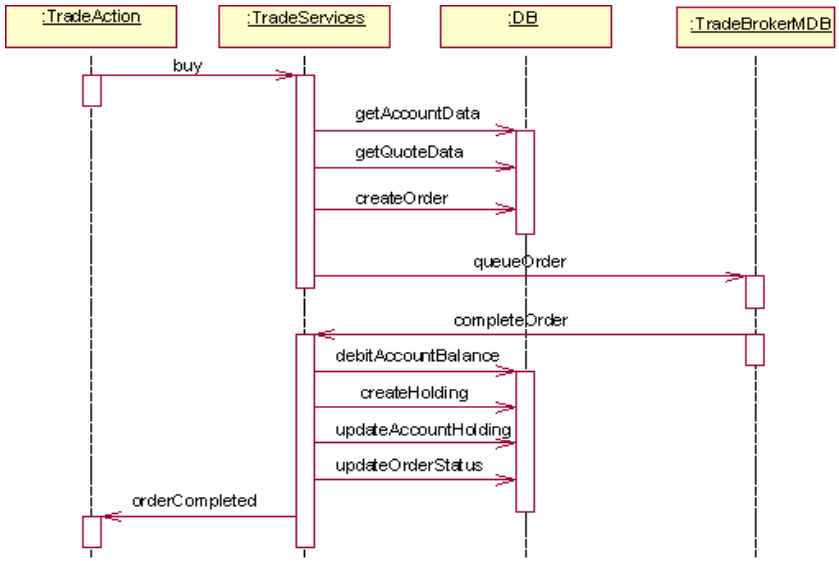

Figure 2. Sequence diagram of buy order.

Figure 2 shows the design model of the use case of buy order. In this design, a participant of type `TradeAction`, which represents the user, sends a buy message to `TradeServices`. `TradeServices` then process the message by communicating to `DB` for retrieving information

of user account and stock quote, and for creating an order. Then it sends a `queueOrder` message to `TradeBrokerMDB`, which represents a trade exchange. After `TradeBrokerMDB` completes the order, it sends a `completeOrder` message back to `TradeServices`. `TradeServices` then updates the user account and status of the order with `DB`, and it sends an `orderCompleted` message to `TradeAction` at the end.

2.2 Relating Models

In this example, we manually create the relationships between the activity diagram (Figure 1) and the sequence diagram (Figure 2) to relate the states and messages (Figure 3). It is noted that an activity can be mapped to a single message or a sequence of messages. This is because a sequence diagram represents a design level model, and so it is a more refined and detailed description of an activity diagram which represents a requirements model. For example, in Figure 3, the activity labeled as s_1 in the activity diagram is mapped to the single message `buy` in the sequence diagram. But activity s_2 is mapped to a sequence of messages `getAccountData`, `getQuoteData`, `createOrder`, and `queueOrder`. Also, it is noted that the order of the activities in the activity diagram matches to the order of the (sequence of) messages in the sequence diagram that corresponds to those activities. This is due to the fact that both diagrams describe the same behavior model of the system, though they are at various levels of abstraction. For example, the activity s_1 in the activity diagram is followed by the activity s_2, and we see that the corresponding `buy` message of s_1 in the sequence diagram is followed by the corresponding sequence of messages (`getAccountData`, `getQuoteData`, `createOrder`, and `queueOrder`) of s_2.

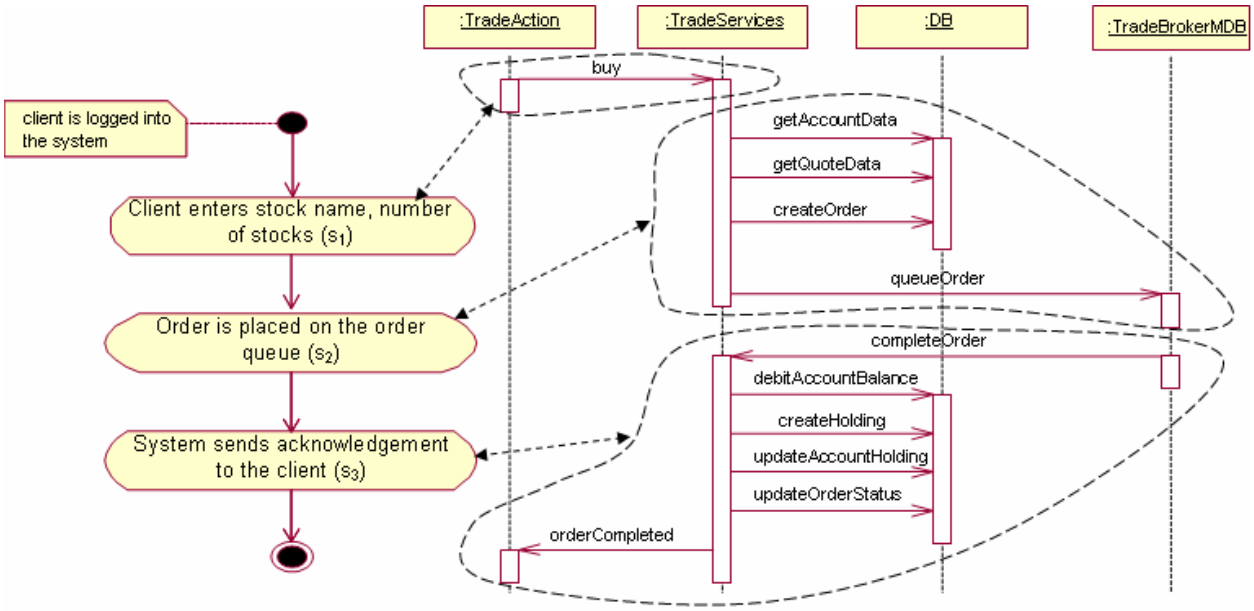

Figure 3. Relates states in activity diagram to messages in sequence diagram.

2.3 Utilizing Relationships for Change Propagation

Using the example, we demonstrate how we use the inter-model relationships to help propagate changes made to one of the models. Suppose the stakeholders of Trade6 request to enhance the buy order use case such that a given order expiration time is checked before the order is executed. As shown in Figure 4, a new activity diagram for the enhanced use case is created to show the change in requirements. In particular, a check of time-limit (c_1) is added to the activity diagram. If the order is within the time-limit, it is executed, and the system sends an acknowledgement to the client (s_3); otherwise, an `order expired` message is sent to the client (s_4). This is a conditional addition to the original activity diagram of buy order.

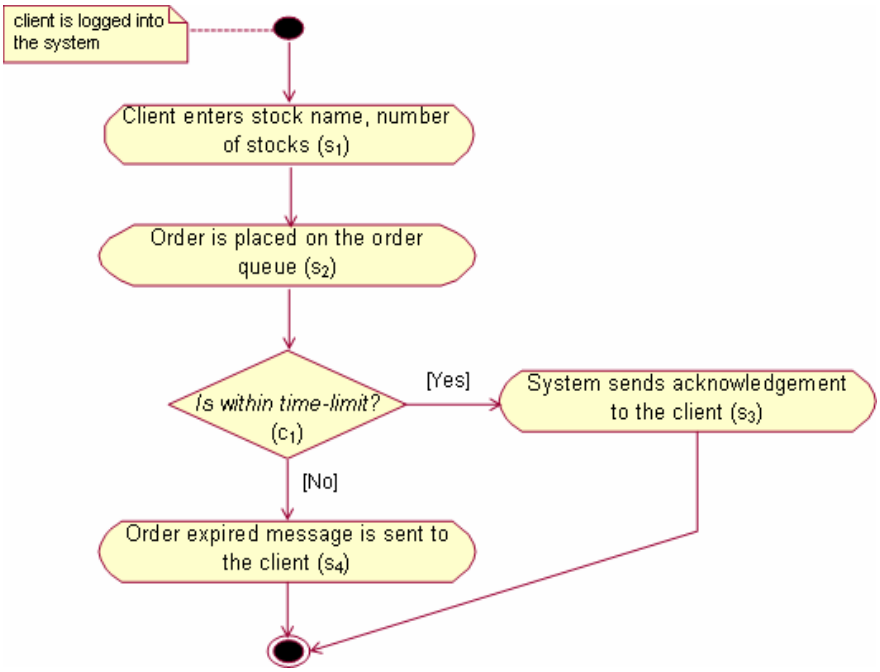

Figure 4. Activity diagram of enhanced buy order

As requirements change, the corresponding designs then need to evolve to accommodate the changes. In the enhanced buy order use case, a new sequence diagram is created as shown in Figure 5 to reflect the corresponding changes in the activity diagram (Figure 4). In particular, a combined alternate interaction fragment is added. If the `within time-limit` constraint is satisfied (the upper fragment), the `completeOrder` message is sent, and `TradeServices` responds by updating the user account and status of the order with `DB`, and finally sends an `orderCompleted` message to `TradeAction`. If the constraint is not met (the lower fragment), an `orderExpired` message is sent and `TradeServices` responds by sending an `orderExpired` message to `TradeAction`.

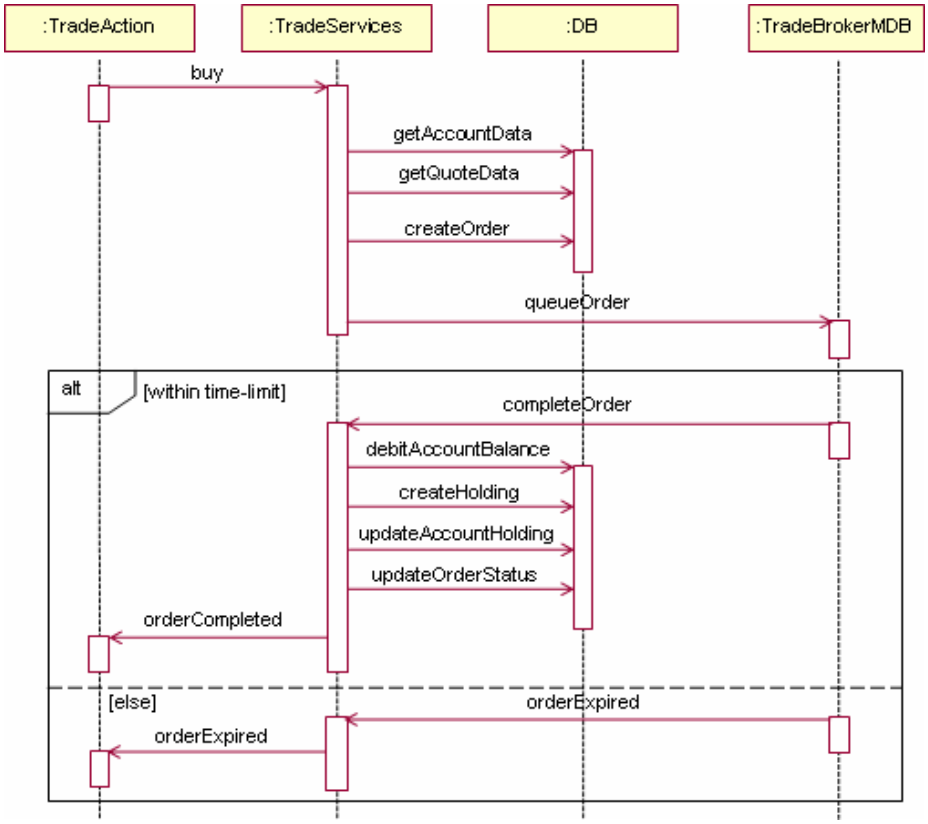

Figure 5. Sequence diagram of enhanced buy order

This example is small and straightforward, and it would be easy to manually identify new changes in sequence diagram. However, a software system in reality is at different scales, and changes in requirements models are not always straightforward. Thus, identifying and making changes in the corresponding design models is error-prone. Thus, we need tools to automatically identify the changes in one model, and provide assistance to the users for propagating the corresponding changes to the more refined models.

As we have established relationships between requirements models and design models, our goal is to utilize these relations to help users in evolving designs. This idea is illustrated in Figure 6. The left-hand side represents the original buy order use case, with the top left diagram showing its activity diagram (ADv1) and the lower left diagram showing its sequence diagram

(SDv1). The relationships between ADv1 and SDv1 have been described in Section 2.2. The right-hand side represents the enhanced buy order use case mentioned above. The top right diagram is its activity diagram (ADv2) and has been given in Figure 4, while the lower right diagram is its sequence diagram (SDv2) and has been shown in Figure 5. Since the new activity diagram ADv2 is evolved from the original activity diagram ADv1, we can identify the activities that are found in both diagrams, and identify the new activities that are added to ADv2. In particular, the pre-condition and the first two activities (s_1 and s_2) are the same in both diagrams. The activity "System sends acknowledgement to the clients" (s_3) is found in both ADv1 and ADv2 but it has different predecessors in the two diagrams. In addition, ADv2 has added a condition check of "Is within time-limit" (c_1) and a new activity "order expired message is sent to the client" (s_4) to one of the branches of the condition check. For the activities that are not changed, the corresponding messages in the original sequence diagram need to be preserved in the new sequence diagram. For example, the first five messages (`buy`, `getAccountData`, `getQuoteData`, `createOrder` and `queueOrder`) in SDv1 are preserved in SDv2. For the relocated activities, the corresponding messages in the original sequence diagram need to be in the new sequence diagram but may appear in different locations. For example, the activity s_3 in ADv1, its corresponding messages (`completeOrder`, `debitAccountBalance`, `createHolding`, `updateAccountHolding`, `updateOrderStatus` and `orderCompleted`) in SDv1 need to appear in SDv2, but they may appear in different locations in the new sequence diagram in order to reflect the change of the predecessor of the activity in the new activity diagram. In addition, the condition check c_1 and the new activity s_4 in ADv2 are then needed to appear in SDv2.

One of the challenges in impact analysis on heterogeneous models is that the domain-specific semantics of each element in an abstraction model is usually not explicitly and directly expressed at the elements in a refined model. The reason that we know s_1 is related to `buy` in the example because we understand those English words in the diagrams. For the sake of propagating changes, we assume that the relations between the original activity diagram and its sequence diagram are granted. From the example, we note that we need to detect changes made in activity diagrams. Specifically, we need to identify what activities are preserved, what are

added to and removed from the original diagram. Also, we need to detect any changes in the ordering of activities because those changes must be reflected in the corresponding sequence diagram. In particular, we need to establish a rule to check the ordering of activities against the ordering of messages. We also note that one activity may correspond to one message or a sequence of messages, so we need to define a notion that captures this relation. At the end of change propagation, we want to identify and localize the impact on sequence diagram.

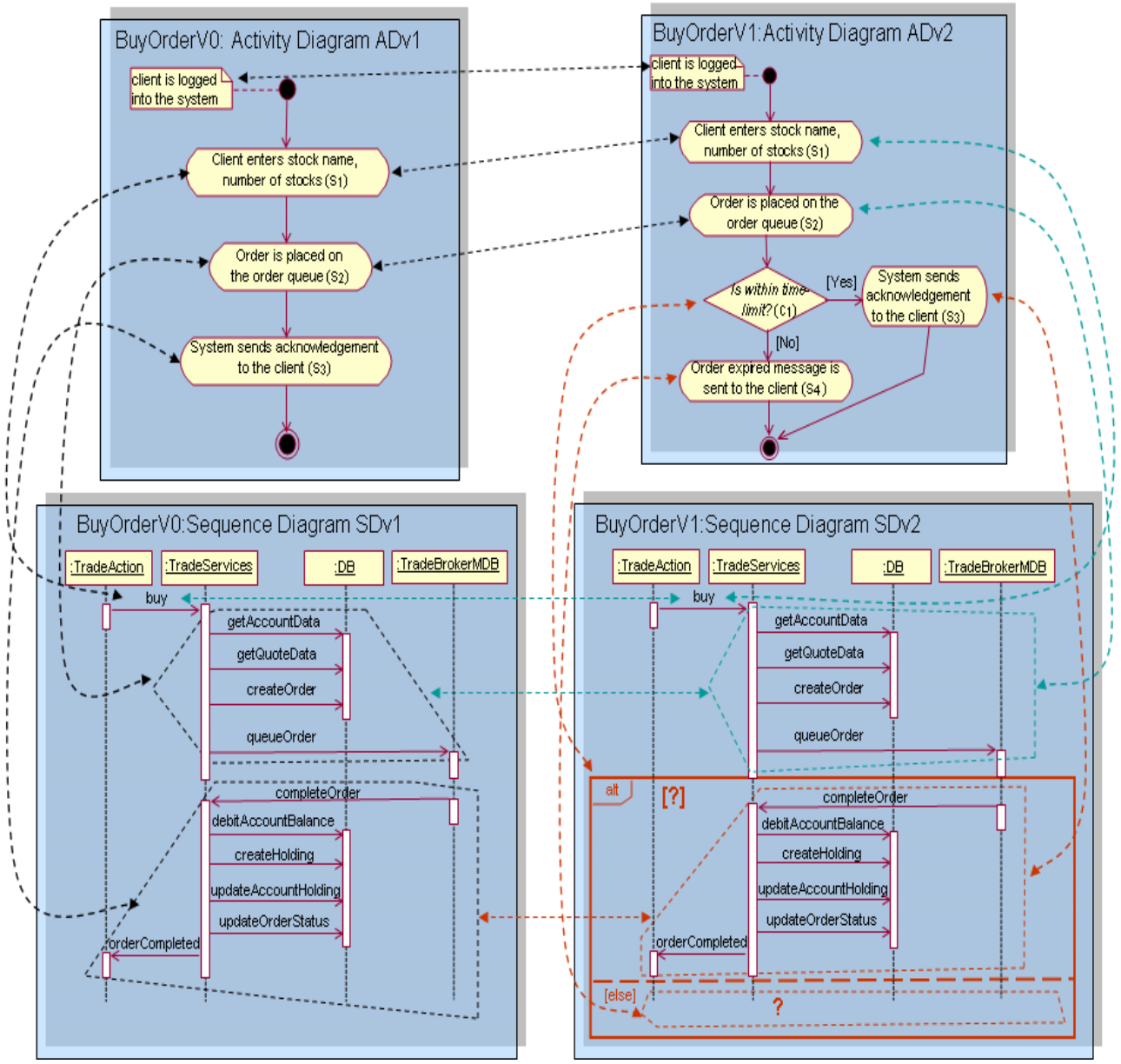

Figure 6. Using relations to propagate changes. The left-hand side is the activity diagram and sequence diagram of the original buy order, and the right-hand sides is the diagrams of the enhanced buy order. Relations between these diagrams are indicated by dashed-arrows.

3 Formalization of Relationships

In this chapter, we formalize models and their relationships. We start with formalizing simple activity and sequence diagrams at the model level. Then we define a conceptual enclosure of messages over a sequence diagram called *region*. Finally, we capture the relationships between activity and sequence diagrams in terms of regions.

Definition 1 (Activity diagram): An *activity diagram* is a tuple $AD = (S, s_0, \Delta, \tau, f_0)$, where S is a set of states, $s_0 \in S$ is an initial state, and $f_0 \in S$ is a final state. τ is a set of transition labels, and we use $\not b$ to denote a blank label. $\Delta \subseteq (S \times \tau \times S)$ is a transition relation relating states and labels. Further, $S = S_i \cup S_d$, where S_i is a set of state activities, and S_d is a set of decision states.

In our example, activity diagram ADv1 of buy order (see Figure 1), S_i is $S = \{s_1, s_2, s_3\}$, τ is $\{\not b\}$, and Δ is $\{(s_0, \not b, s_1), (s_1, \not b, s_2), (s_2, \not b, s_3), (s_3, \not b, f_0)\}$.

Definition 2 (Sequence diagram): A *sequence diagram* is a tuple $SD = (E, L, I, M, instance, order, label)$, where E is a set of events, partitioned into $send(E)$ and $receive(E)$, where $send(E)$ denotes a set of send events and $receive(E)$ denotes a set of receive events. L is a set of message labels. I is a set of instances. M is a bijection: $send(E) \rightarrow receive(E)$. The function $instance : E \rightarrow I$ maps an event to the instance in which the event occurs. The function $label : E \rightarrow L$ maps events to labels. *Order* is a set of total orders $\{\leq_i \mid i \in I\}$, and $\leq_i \subseteq \{ e \in E \mid instance(e) = i \} \times \{ e \in E \mid instance(e) = i \}$ denotes the total order of events on instance i.

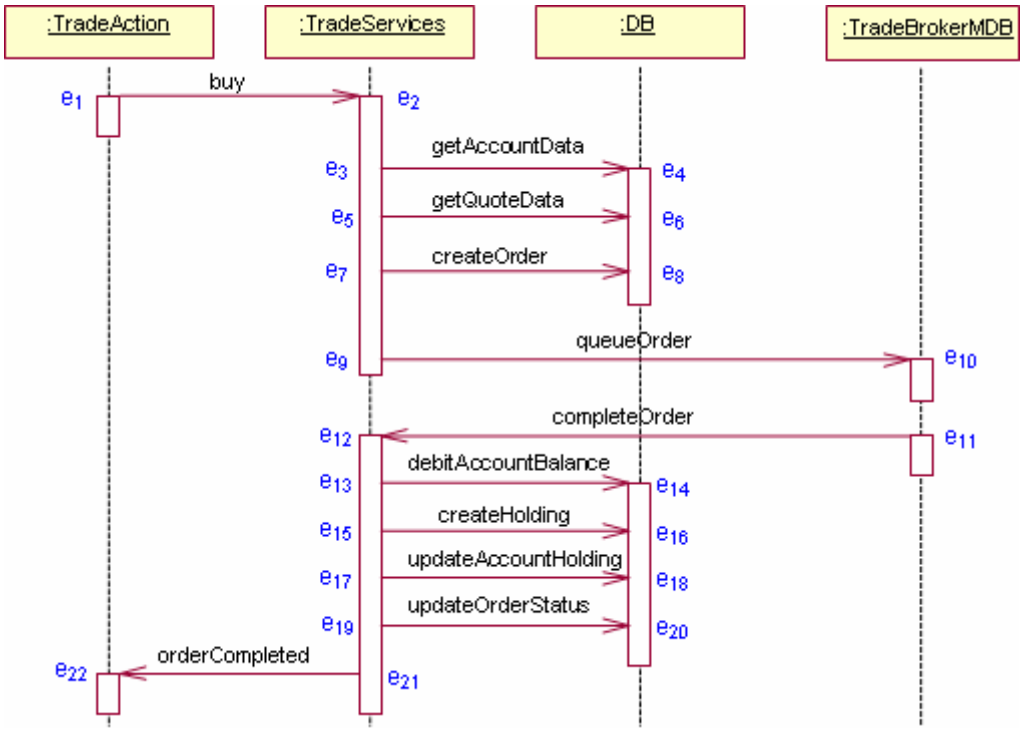

Figure 7. Sequence diagram of buy order. The e_i's are added to show the occurrences of the events of the corresponding message.

An example of a sequence diagram SDv1 is shown in Figure 7. The set of instances I is $\{$`TradeAction, TradeServies, DB, TradeBrokerMDB`$\}$. L is the set of labels shown above the message arrows $\rightarrow$ or $\leftarrow$, e.g., `buy`, `getQuoteData`, `orderCompleted`. The events E is $\{e_1, e_2, \ldots, e_{22}\}$, where $send(E)$ is $\{e_1, e_3, e_5, \ldots, e_{21}\}$, and $receive(E)$ is $\{e_2, e_4, e_6 \ldots, e_{22}\}$. An example of bijection M is $M(e_1)$ returns e_2. The function $instance(e_1)$ returns `TradeAction`, and $instance(e_2)$ returns `TradeServices`. Both $label(e_1)$ and $label(e_2)$ are `buy`. The total order of events on instance `TradeAction`, denoted as $\leq_{TradeAction}$, is $\{e_1 \leq_{TradeAction} e_{22}\}$. The set of total orders *order* is the union of $\leq_{TradeAction}$, $\leq_{TradeServices}$, $\leq_{DB}$, and $\leq_{TradeBrokderMDB}$, where the last three denote the total order of events on instances `TradeServices`, `DB`, and `TradeBrokerMDB`, respectively.

Definition 3 (Partial order over events): Given a sequence diagram SD, $\preceq_{SD} \subseteq E \times E$ denotes partial ordering of events in SD so that $\preceq_{SD} = \bigcup_{i \in I} \leq_i \cup \{(e, M(e)) \mid e \in send(E)\}$. $e_1 \preceq_{SD} e_2$ holds if e_1 happens before e_2. Also, $(e_1 \preceq_{SD} e_2) \wedge (e_2 \preceq_{SD} e_3) \Rightarrow e_1 \preceq_{SD} e_3$.

In the example SDv1 (Figure 7), the partial order over events in the sequence diagram $\preceq_{SD}$ is a union of $\leq_{TradeAction}$, $\leq_{TradeServices}$, $\leq_{DB}$, $\leq_{TradeBrokderMDB}$, and

$$\{(e, M(e)) \mid e \in send(E)\} = \{e_1 \preceq_{SD} e_2, e_3 \preceq_{SD} e_4, \ldots, e_{21} \preceq_{SD} e_{22}\}.$$

Also, we know $e_1 \preceq_{SD} e_3$ because $e_1 \preceq_{SD} e_2$ and $e_2 \preceq_{SD} e_3$ as the latter is in $\leq_{TradeServices}$.

Definition 4 (Region): Suppose we are given a sequence diagram SD and its partial ordering over events $\preceq_{SD}$. Let $sd' = (E', L', I', M', instance', label', order')$ be a tuple and $\preceq_{sd'}$ be partial ordering over its events E'. The tuple sd' is a *region* of SD if all the following properties hold:

(a) $E' \subseteq E$, $L' \subseteq L$, $I' \subseteq I$, $M' \subseteq M$, $instance' \subseteq instance$, $label' \subseteq label$, $order' \subseteq order$, and $\preceq_{sd'} \subseteq \preceq_{SD}$.

(b) $\forall x, y \in E'.\ x \preceq_{sd'} y \Rightarrow (\forall z \in E\ (x \preceq_{SD} z \wedge z \preceq_{SD} y \Rightarrow z \in E'))$.

(c) $\forall x.\ min_{sd'}(x) \Rightarrow x \in send(E')$. (Refer to Definition 5 for the description of $min_{sd'}$)

(d) $\forall x.\ max_{sd'}(x) \Rightarrow x \in receive(E')$. (Refer to Definition 6 for the description of $max_{sd'}$)

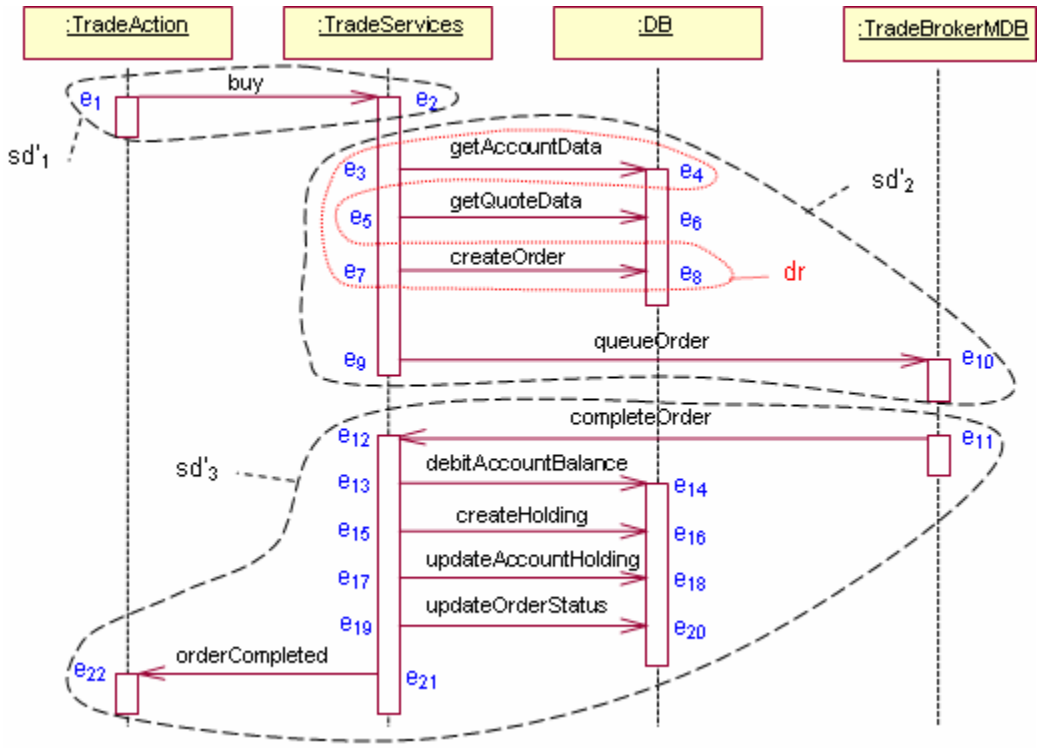

Figure 8. Sequence diagram of buy order. The black dashed areas denoted by sd'$_i$ **are valid regions, while the dotted area denoted by** dr **is an example of an invalid region.**

Figure 8 illustrates the idea of regions in SDv1. The tuple

$$sd'_2 = (E'_2, L'_2, I'_2, M'_2, instance'_2, label'_2, order'_2)$$

is an example of a region over the sequence diagram. Its events E'_2 is $\{e_3, e_4, \ldots, e_{9,} e_{10}\}$, its set of labels L'_2 is $\{$ `getAccountData, getQuoteData, createOrder, queueOrder` $\}$, the instances of the region I'_2 is $\{$ `TradeServices, DB, TradeBrokderMDB` $\}$, and the events bijection M'_2 is $\{ e_3 \rightarrow e_4, e_5 \rightarrow e_6, e_7 \rightarrow e_8, e_9 \rightarrow e_{10} \}$. The function $instance'_2(e_3)$ returns `TradeServices`, and $instance'_2(e_4)$ returns `DB`. Both $label'_2(e_3)$ and $label'_2(e_4)$ are `getAccountData`. The total order of events on instance `TradeServices`, denoted as

$\leq'_{2TradeServices}$, is a set of tuples $\{(e_3,e_5),(e_3,e_7),(e_3,e_9),(e_5,e_7),(e_5,e_9),(e_7,e_9)\}$. As shown in Figure 9, the partial order over events in the region $\preceq_{sd'_2}$ is jointly formed from $\{(e,M'_2(e)\,|\,e\in send(E'_2)\}$, $\leq'_{2TradeServices}$, $\leq'_{2DB}$, and $\leq'_{2TradeBrokerMDB}$. For example, event e_3 happens before e_4 and e_5, and there is no order between e_4 and e_5.

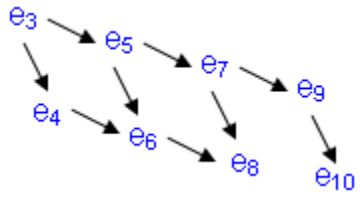

Figure 9. Partial order of events over a region sd'$_2$ of SDv1 shown in Figure 8. The notation $e_x \rightarrow e_y$ means that e_x happens before e_y.

Definition 5 (Minimum event predicate): Given a region sd', predicate $min_{sd'} \subset E'$ denotes minima of $\preceq_{sd'}$ so that $min_{sd'}(x)$ holds if $\nexists y \in E'$ s.t. $y \preceq_{sd'} x$.

Definition 6 (Maximum event predicate): Given a region sd', predicate $max_{sd'} \subset E'$ denotes maxima of $\preceq_{sd'}$ so that $max_{sd'}(x)$ holds if $\nexists y \in E'$ s.t. $x \preceq_{sd'} y$.

From the partial order over events of sd'_2 (Figure 9), the minimum event predicate $min_{sd'_2}(x)$ is true for e_3, which is a send event; while the predicate is false for other events. The maximum event predicate $max_{sd'_2}(x)$ is true only for e_8 and e_{10}, which are receive events. This also illustrates that a region can have more than one minimum and one maximum event. We also know that sd'_2 in Figure 8 is a region because it satisfies all the properties of Definition 4.

In Figure 8, a tuple representing the area dr enclosed by the dotted line is not a region because property (b) of Definition 4 is unsatisfied. In particular, $e_5 \preceq_{sd'} e_9$ requires that e_7 be in

the events of this tuple in order to satisfy the property. Further, if e_7 is added to the events, e_8 needs to be added as well because the mapping between send and receive events is a bijection.

Definition 7 (Partial ordering over regions): Let $R = \{sd'_1, \ldots, sd'_n\}$ be a set of regions of a sequence diagram SD, and for each region sd'_i, its partial ordering over events be $\preceq_{sd'_i}$. Let $\preceq_R \subseteq R \times R$ denote a partial ordering over regions. $r_i \preceq_R r_j$ if $\neg(\exists x \exists y . max_i(x) \wedge min_j(y) \wedge y \preceq_{SD} x) \wedge (\exists x \exists y . max_i(x) \wedge min_j(y) \wedge x \preceq_{SD} y)$. In other words, given two regions r_i and r_j, we say r_i happens before r_j if (i) all the minimum events of r_j do not happen before any maximum event of r_i and (ii) some maximum events of r_i happen before some minimum events of r_j.

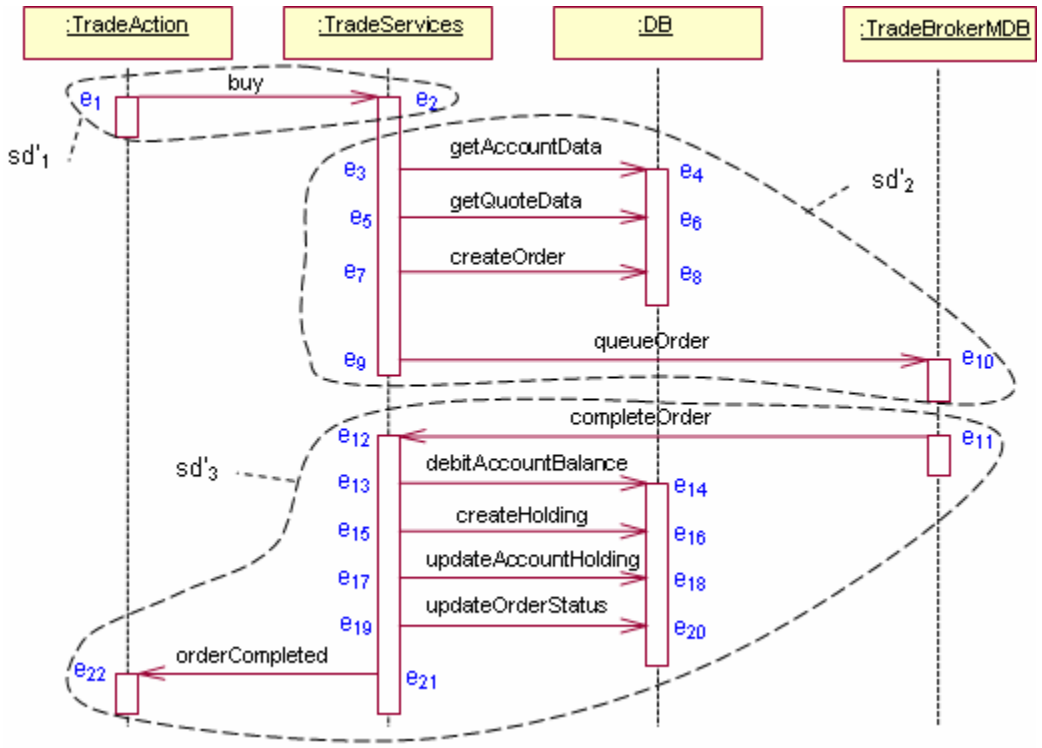

Figure 10. Sequence diagram of buy order, with regions

Figure 10 illustrates the order over the set of regions $R = \{ sd'_1, sd'_2, sd'_3 \}$ in SDv1. It is noted that e_2 is the only maximum event of sd'_1, and $e_2 \preceq_{SD} e_3$, so we know $sd'_1 \preceq_R sd'_2$. Also, e_{11} is the only minimum event of sd'_3, $e_{10} \preceq_{SD} e_{11}$, and there is no order between e_8 and e_{11}, so we conclude $sd'_2 \preceq_R sd'_3$.

Definition 8 (Relations between activity diagram and sequence diagram): Let SD be a sequence diagram, R be a set of regions of SD, AD be an activity diagram, and S be the states in AD. Let $\rho \subseteq R \times S$ be a relation between the regions and states. We use this relationship to find whether the ordering over the regions R in the sequence diagram SD has any conflict with the ordering of states S in the activity diagram AD. If a region x happens before a region x', the state y' corresponding to the region x' does not happen before the state y corresponding to the region x. Also, if a state y happens before a state y', the region x' corresponding to the state y' does not happen before the region x corresponding to the state y. Formally,

$$\forall x \forall y (x, y) \in \rho \wedge \forall x' \forall y' (x', y') \in \rho,\ (x \preceq_R x' \Rightarrow \forall z (y', z, y) \notin \Delta) \wedge (\exists z (y, z, y') \in \Delta \Rightarrow x' \not\preceq_R x).$$

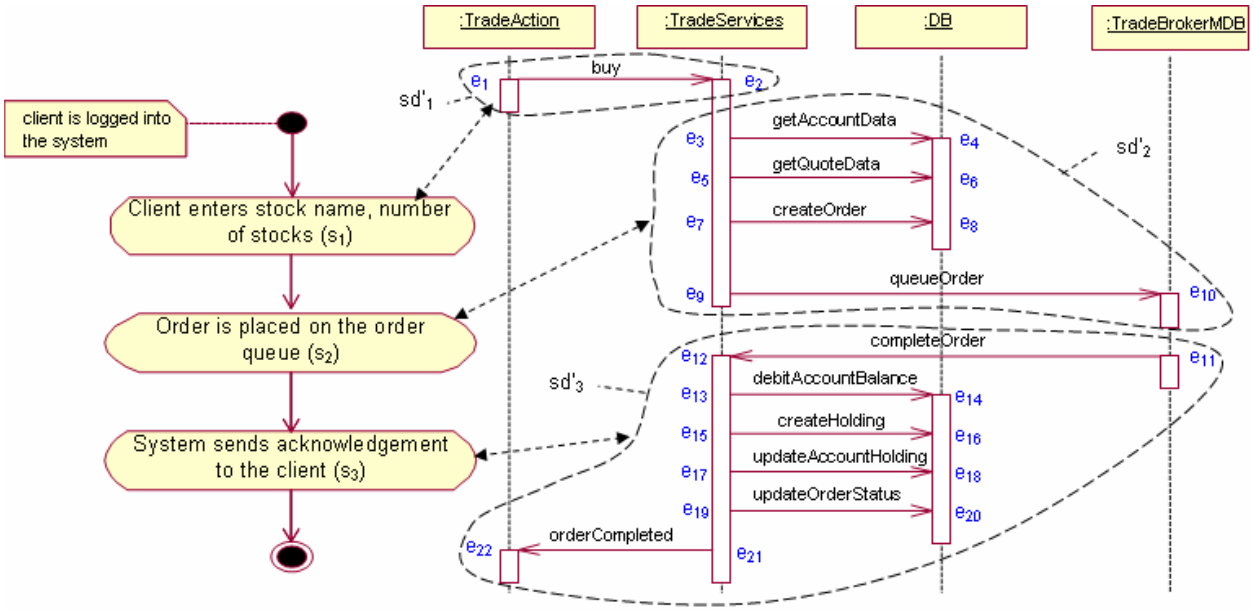

Figure 11. Relations between activity states and regions.

Figure 11 illustrates relations between activity states and regions. Suppose we are given a relation $\rho = \{(sd'_1, s_1), (sd'_2, s_2), (sd'_3, s_3)\}$. We know s_1 relates to sd'_1, and s_2 corresponds to sd'_2. Also, we know that $sd'_1 \preceq_R sd'_2$ from Definition 7. Hence by Definition 8, we know there should be no transition from s_2 to s_1, and this is true by inspecting the activity diagram. On the other hand, the activity diagram indicates that there is a transition from s_1 to s_2, and so we expect sd'_2 does not happen before sd'_1. This is confirmed in the sequence diagram. Suppose now we are given a subset $\rho_0 = \{(sd'_1, s_1), (sd'_2, s_2)\}$, and we want to discover any missing relations. From the given ρ_0, we know that s_2 relates to sd'_2. Also, we know that $sd'_2 \preceq_R sd'_3$ from Definition 7. The activity diagram indicates that there is a transition from s_2 to s_3. It is safe to say that any intermediate activities right after s_2 are related to any intermediate regions after sd'_2. If there is only a single activity after s_2 and only a single region after sd'_2, then we can conclude that this single activity (s_3) relates to this single region (sd'_3).

4 Automating Change Propagation

In this chapter, we show how to automatically propagate a change in an abstraction model to a refined model by using the inter-model relationships defined in Chapter 3. We start with formalizing changes made in an activity diagram (Section 4.1), then we give an algorithm that propagates the changes to the corresponding sequence diagram (Section 4.2), and illustrate it with the buy order use case (Section 4.3). We wrap this chapter by discussing possible extensions of our model and algorithm, and challenges of full automation.

4.1 A Notion of Change

Suppose we are given a version of an activity diagram AD_1 and its sequence diagram SD_1. Let S_1 be the states in AD_1, and R_1 be the regions in SD_1. Further, we are given the relation between R_1 and S_1, namely, ρ_1. Also, we are given a new version of activity diagram AD_2. In addition, we are given ρ_{AD} that relates the states between S_1 and S_2 (states of AD_2). We want to locate the changes needed in the sequence diagram corresponding to AD_2, and we do this in an algorithm *LocateChange*. Figure 12 gives a pictorial description of this algorithm. It takes AD_1, SD_1, ρ_1, AD_2 and ρ_{AD} as its input, and generates a new sequence diagram SD_2 with placeholders for the potential places of new regions that correspond to the new activities in AD_2.

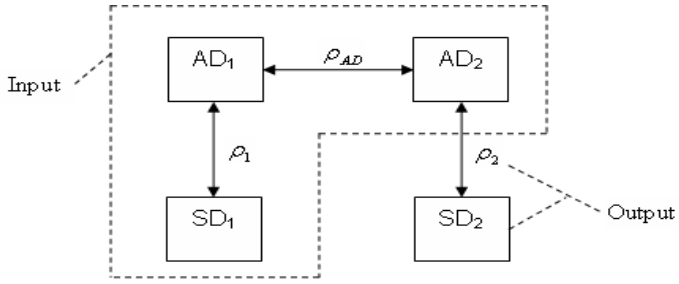

Figure 12. Input and output of algorithm *LocateChange*

Before we give the algorithm *LocateChange*, we formalize a function *NotCovered* that is used in the algorithm for determining the states that are added or removed from activity diagrams.

Definition 9 (function *NotCovered*): Let AD_1 be the original version of the activity diagram, S_1 be the states of AD_1, and Δ_1 be the transition relation of AD_1. Also, let AD_2 be the new version of the activity diagram, S_2 be the states of AD_2, and Δ_2 be the transition relation of AD_2. Suppose we are given a relation $\rho_{AD} \subseteq S_1 \times S_2$ that relates the states between the two activity diagrams. We define a function *NotCovered* that takes ρ_{AD} and S_1 (respectively S_2) as input and returns the states in S_1 (respectively S_2) that are not covered by the relations between S_1 and S_2. Formally, $NotCovered(\rho_{AD}, S_1) = \{s \in S_1 \mid \nexists s' \in S_2.(s, s') \in \rho_{AD}\}$, and $NotCovered(\rho_{AD}, S_2) = \{s \in S_2 \mid \nexists s' \in S_1.(s', s) \in \rho_{AD}\}$.

To elaborate a bit, if $NotCovered(\rho_{AD}, S_1)$ returns an empty set, then the states in the activity diagram AD_1 are not removed. Otherwise, the non-empty set specifies the states that are removed from AD_1. Similarly, if $NotCovered(\rho_{AD}, S_2)$ returns an empty set, then no state is added to the activity diagram AD_2. Otherwise, the non-empty set tells the states that are added

to AD_2. In the example given in Figure 6, let S_1 be the states of the activity diagram ADv1, S_2 be the states of the activity diagram ADv2, and ρ_{AD} be the relations between the states in ADv1 and ADv2. The function $NotCovered(\rho_{AD}, S_1)$ returns an empty set and so no state in ADv1 is removed. In contrast, the function $NotCovered(\rho_{AD}, S_2)$ returns two states, namely, the condition check c_1 and the activity s_4, indicating that c_1 and s_4 are the states added to ADv2.

4.2 Algorithm

We are now ready to describe the algorithm *LocateChange* (Figure 13). The algorithm starts by finding the difference of states between AD_1 and AD_2. In particular, it finds the states that are added to AD_2, namely, *addedStates*, and the states that are removed from AD_1, namely, *removedStates*. It then initializes the new sequence diagram SD_2 for AD_2 by copying the regions from SD_1 whose corresponding states are not in *removedStates*. It also initializes the relations ρ_2 between the regions in SD_2 and the states S_2 in AD_1. This is done by first taking the regions just established in SD_2 (which are the cloned regions in SD_1), and then finding the corresponding states in AD_1 through the relations ρ_1, and finally using the relations ρ_{AD} to find the states in AD_2 that are mapped to those states we just found in AD_1. After these initializations, the algorithm iterates over every state in the *addedStates* by finding the placeholders of regions in SD_2 that correspond to these new states. In particular, for a given new state y, the algorithm finds its predecessor (or its successor) x in AD_2 and looks for the state x_1 in AD_1 that is related

to x. Then it finds the region sd_1 in SD_1 that is related to x_1. If the region sd_1 can be found, a placeholder is inserted after (or before) sd_1 in SD_2 as a potential place for the region sd in SD_2 that corresponds to the state y in AD_2. The relationship ρ_2 is also updated with the relation between sd and y. If the state x_1 in AD_1 cannot be found, it means that the predecessor (or the successor) x of the new state y is also a new state in AD_2. In this case, we do not explicitly add a dedicated region for y in SD_2; instead we extend the placeholder of the region for x in SD_2 to hold the messages corresponding to y as well. Finally, the algorithm checks for any potential violation of ordering and reports them to users for manual fix. In particular, by using ρ_{AD}, we look for any state y in AD_2 and its related state s_1 in AD_1, such that the predecessor of y is not related to the predecessor s_1. Also, for every state y and its predecessor x in AD_2, we find the region sd' of y and the region sd of x in SD_2 by using ρ_2, and then we check whether the ordering between x and y is in conflict with the ordering between sd and sd'.

LocateChange
input:
AD_1, // activity diagram version 1
SD_1, // sequence diagram version 1 for AD_1
ρ_1, // relations between regions in SD_1 and states in AD_1
AD_2, // activity diagram version 2
ρ_{AD} // relations between states in AD_1 and AD_2
output:
SD_2, // sequence diagram version 2 for AD_2
ρ_2 // relations between regions in SD_2 and states in AD_2

begin
// Let S_2 = states in AD_2, find the states added to AD_2
$addedStates = NotCovered(\rho_{AD}, S_2)$
// Let S_1 = states in AD_1, find the states removed from AD_1.

$removedStates = NotCovered(\rho_{AD}, S_1)$.

// Initialize SD_2 with SD_1, but only keep regions whose corresponding states are in AD_2.
$SD_2 = \{sd \mid sd \in SD_1 \wedge (sd, s) \in \rho_1 \wedge s \notin removedStates\}$
// Initialize ρ_2 with the relations between the regions that are just put in SD_2
// and the states in AD_2 that correspond to those regions.
$\rho_2 = \{(sd, s') \mid sd \in SD_2 \wedge \exists s(\ (sd, s) \in \rho_1 \wedge (s, s') \in \rho_{AD}\ )\}$

// Iterate over every new state in AD_2
for each new state y **in** *addedStates*
  // Let sd denote the new region we need to add in SD_2 for y
  // if we can find a region sd_1 in SD_1 such that
  // (i) sd_1 is related to state x_1 in AD_1,
  // (ii) x_1 is related to x in AD_2, and
  // (iii) x is a predecessor of the new state y in AD_2
  if $\exists x \exists u(x, u, y) \in \Delta_2$ **&&** $\exists x_1(x_1, x) \in \rho_{AD}$ **&&** $\exists sd_1(sd_1, x_1) \in \rho_1$ **then**
    insert placeholder for region sd after a region equivalent to sd_1 in SD_2
    add (sd, y) to ρ_2
  // or, if we can find a region sd_1 in SD_1 such that
  // (i) sd_1 is related to state x_1 in AD_1,
  // (ii) x_1 is related to x in AD_2, and
  // (iii) x is a successor of the new state y in AD_2
  else if $\exists x \exists u(y, u, x) \in \Delta_2$ **&&** $\exists x_1(x_1, x) \in \rho_{AD}$ **&&** $\exists sd_1(sd_1, x_1) \in \rho_1$ **then**
    insert placeholder for region sd before a region equivalent to sd_1 in SD_2
    add (sd, y) to ρ_2
  endif
end for

// Iterate the states S_2 in AD_2 to check for ordering violations
for each state y in S_2
  Let x = predecessor of y
  Let s_1 = related state of y in AD_1
  Let x_1 = predecessor of s_1
  Let sd = region of x in SD_2
  Let sd' = region of y in SD_2
  if $(x_1, x) \notin \rho_{AD}$ **or** ordering of *x* and *y* violates ordering of sd and sd' **then**
    print sd' as potential violations for users to fix
  endif

```
  end for
end
```

Figure 13. Algorithm for locating changes.

We warp up this section by discussing the complexity of the algorithm. The complexity of $NotCovered(\rho_{AD}, S)$ is $O(|S| \times |\rho_{AD}|)$, SD_2 initialization is $O(|R_1| \times (|\rho_1| + |removedStates|))$, and ρ_2 initialization is $O(|R_2| \times (|\rho_1| + |\rho_{AD}|))$, where R_1 (respectively R_1) is the regions in SD_1 (respectively SD_2). For every new state y in AD_2 , the complexity of locating placeholders for its regions is $O(|ps_y| \times (|\rho_{AD}| + |\rho_1| + |\rho_2|))$, where ps_y is the predecessors (or successors) of y . Thus, it takes $O(|S_2| \times |ps_y| \times (|\rho_{AD}| + |\rho_1| + |\rho_2|))$ to locate regions for new states in AD_2 . For checking against ordering violations of states in activity diagrams, the complexity is $O(|S_2| \times |ps_y| \times |\rho_{AD}|)$. To check against ordering violations of regions, we refer to partial ordering over regions (Definition 7). We need to find all the minimum and maximum events of all regions, and the complexity for each region r is $O(|\preceq_r|)$. Let $maxE_{sd}$ denote the maximum events of region sd , and $minE_{sd'}$ denote the minimum events of region sd' . The complexity of checking the ordering of two regions $sd \preceq_R sd'$ is $O(|maxE_{sd}| \times |minE_{sd'}| \times |\preceq_{SD}|)$. Thus, the complexity of checking against ordering violations of regions is $O(|S_2| \times |ps_y| \times (|\rho_2| + |\preceq_r| + |maxE_{sd}| \times |minE_{sd'}| \times |\preceq_{SD}|))$. Note that $|\rho_1|$, $|removedStates|$ and $|R_1|$ are bounded by $|S_1|$, $|R_2|$, $|\rho_2|$ and $|ps_y|$ are bounded by $|S_2|$, and $|\rho_{AD}|$ is bounded by $\min(|S_1|, |S_2|)$. Also, $|maxE_{sd}|$ and $|minE_{sd'}|$ are bounded by $|E|$, and $|\preceq_r|$ and $|\preceq_{SD}|$ are bounded

by $|E|^2$, where E is the events of SD_1. Thus the complexity of the algorithm is $O(|S_2|^3 + |S_2|^2|E|^4)$. In practice, $\preceq_{SD}$ can be stored in a hash table, making the expected running time closer to $O(|S_2|^3 + |S_2|^2|E|^2)$.

4.3 Illustration

We illustrate the algorithm using the example given in Figure 6. First, *addedStates* is $\{c_1, s_4\}$, *removedStates* is an empty set, and so $\{s_1, s_2, s_3\}$ are preserved in both AD_1 and AD_2. The sequence diagram SD_2 is initialized with the regions in SD_1 that correspond to s_1, s_2 and s_3. Also, the relationship ρ_2 is initialized with the relations between the regions in SD_2 and the corresponding states s_1, s_2 and s_3 in AD_2. For the new state c_1, its predecessor in AD_2 is s_2. The state s_2 in AD_2 is mapped to the state s_2 in AD_1, and the state s_2 in AD_1 is related to the region sd_2 (`getAccountData`, `getQuoteData`, `createOrder`, `queueOrder`) in SD_1. So a placeholder for the region corresponding to the state c_1 is inserted after the region sd_2 in SD_2. For the new state s_4, its predecessor in AD_2 is c_1. Since c_1 is not found in the relations ρ_{AD}, we assume that the placeholder for the region of c_1 in SD_2 is to be extended to hold the messages of s_4. After handling the new states, we check all the states in AD_2 for any violation of ordering. The predecessor of the state s_3 in AD_2 is c_1, while the predecessor of s_3 in AD_1 is s_2. Since s_2 and c_1 are not related, we report this violation for manual inspection. The predecessor of c_1 in AD_2 is s_2; in contrast, the predecessor of c_1 in AD_1 does not exist because c_1 is a new state. Since the two predecessors cannot be related, we report the placeholder for the

region of c_1 in SD_2 for manual fix. Similarly, we report the placeholder for the region of s_4 for manual inspection because s_4 is a new state.

4.4 Discussion

In what follows, we discuss two possible extensions of our model given in Chapter 3 and the algorithm in this chapter, and challenges of full automation.

Typed states

The meta-model of activity diagram [12] defines a type `ActivityNode` and further extends the type to a derived type `Action` for representing an action in the diagram, and a derived type `DecisionNode` for representing logic controls in the diagram. Although we do not distinguish between these two notions in our model and simply use state to represent them (Definition 1 in Chapter 3), this does not affect the algorithm *LocateChange* (Chapter 4) for propagating changes. This is because we can still use the state transitions to find the predecessor (and the successor) of a state, and use the relations to find the corresponding states in the old activity diagram, and to find the regions of those states in the old sequence diagram. If there is a need to decide the detailed locations of new regions, say, fragments inside a combined alt-fragment in a sequence diagram, we can extend our states to define typed states and use the transition labels between those states to help us deciding the details of a new sequence diagram.

Loops

In the example we study so far, we assume there is only one predecessor for any given state. This is not true for all activity diagrams, for example, we may have a loop as a logic control. To address this, we can extend *LocateChange* to fully utilize the information given by transition labels. To elaborate, if a state has two predecessors, each of the predecessors can be

uniquely identified by the transition label between the predecessor and the state. In the case of loop control, we may end up having a region whose preceding regions are graphically separated. Also, we have a conflict in partial ordering of regions, namely, a region preceding another region ($sd'_2 \preceq_R sd'_1$) also comes after the same region ($sd'_1 \preceq_R sd'_2$). We can report this conflict to the user for manual fix. If we build in typed states, we may realize that the related regions in the original sequence diagram will need to be embedded in a new combined loop fragment in the new sequence diagram.

Challenges of Full Automation

Our algorithm thus far automatically locates the placeholders of regions that correspond to changes in activity diagrams. However, we encounter various challenges of fully automating change propagation. To name a few of them, it is difficult to decide the instances and messages over a new region, and the granularity of messages. In particular, a new region may involve a subset or full set of the participants in the sequence diagram, or new participants that are not yet in the diagram. This is partly because there is a lack of knowledge about the domain of applications, about the design intent of the participants and their responsibilities, and partly because there could be multiple designs that meet the same requirement and so it becomes a subject of using the best design principle. The difficulty of automatically synthesizing messages comes from the fact that the requirements and design models represent two levels of abstraction. A simple description of an activity may be implemented as a simple message or a sequence of synchronous or asynchronous messages across different participants. We have seen activity s_1 maps to a single message in region sd'_1 while activity s_3 is implemented as a sequence of six

messages in region sd'_3 (Figure 11). As an example of the above arguments, the designer wants to create a sequence diagram for the use case of sell order from the existing software artifacts of buy order use case, and he/she starts with changing the activity diagram of buy order. Since we understand the words “buy”, “sell”, and we know the application software should credit the user’s account with the proceeds of the sell order instead of debiting the account when the order is completed, propagating this change to the corresponding sequence diagram looks easy to humans. However, it becomes a challenging task to automated tools because the tools need to interpret those words (with an aid from external tools), and walk through each message in regions to decide whether there is a need of change. As another example, the requirements of the sell order are evolved to support withdrawal of tax from proceeds based on certain taxation rules. It is difficult for tools to automatically recognize that new participants (for example, revenue agency) are needed in the new sequence diagram. The difficulty of deciding the granularity of messages can be seen from adding a requirement to the use case of buy order that allows users to buy stocks from foreign exchanges. Should message `debitAccountBalance` be broken down into a sequence of messages that explicitly take foreign currencies into account? These kinds of challenges cannot be solely solved by model and meta-model, therefore, our algorithm reports regions to designers for manual fix.

5 Towards Tool Support

In this chapter, we describe how the algorithm (Figure 13) can be implemented in existing modeling tools. Since modeling tools like MMTF [13] are based on meta-models, we first briefly describe the meta-models of activity and sequence diagrams [12]. Then we discuss how the algorithm that is based on models and relations can be implemented in the language of these meta-models.

The meta-model of activity diagrams is given on the left-hand side of Figure 14. The `Action` in the meta-model represents an activity, and is a regular state in our model of activity diagram (Definition 1). The `DecisionNode` is a decision node in the diagram, and an example is the diamond shaped condition node. Both `Action` and `DecisionNode` are of type `ActivityNode`. While the meta-model distinguishes between the types `DecisionNode` and `Action`, we treat them the same in our model. The `ActivityEdge` in the meta-model represents a transition label in the diagram. An `ActivityEdge`, a source `ActivityNode`, and a target `ActivityNode` form a transition from a source state to a target state. The meta-model of sequence diagrams is given on the right-hand side of Figure 14. The `InteractionFragment` in the meta-model represents an interaction or a fragment in a sequence diagram. We are interested in fragments as they are similar to the regions (Definition 4) in our model. In particular, the meta-model defines a type `CombinedFragment` parameterized with value `alt` for representing an alternate-fragment in the diagram. The `GeneralOrdering` specifies the ordering of the fragments, and so it can tell the ordering of messages in the sequence diagram. The ordering of these fragments corresponds to the ordering of events (Definition 3) and the

ordering of regions (Definition 7) in our model. The figure also gives the relationships of elements between two meta-models, in particular, it relates `DecisionNode` to `CombinedFragement`, `ActivityNode` to `InteractionFragment`, and the transition formed by `ActivityNode` and `ActivityEdge` to `GeneralOrdering`.

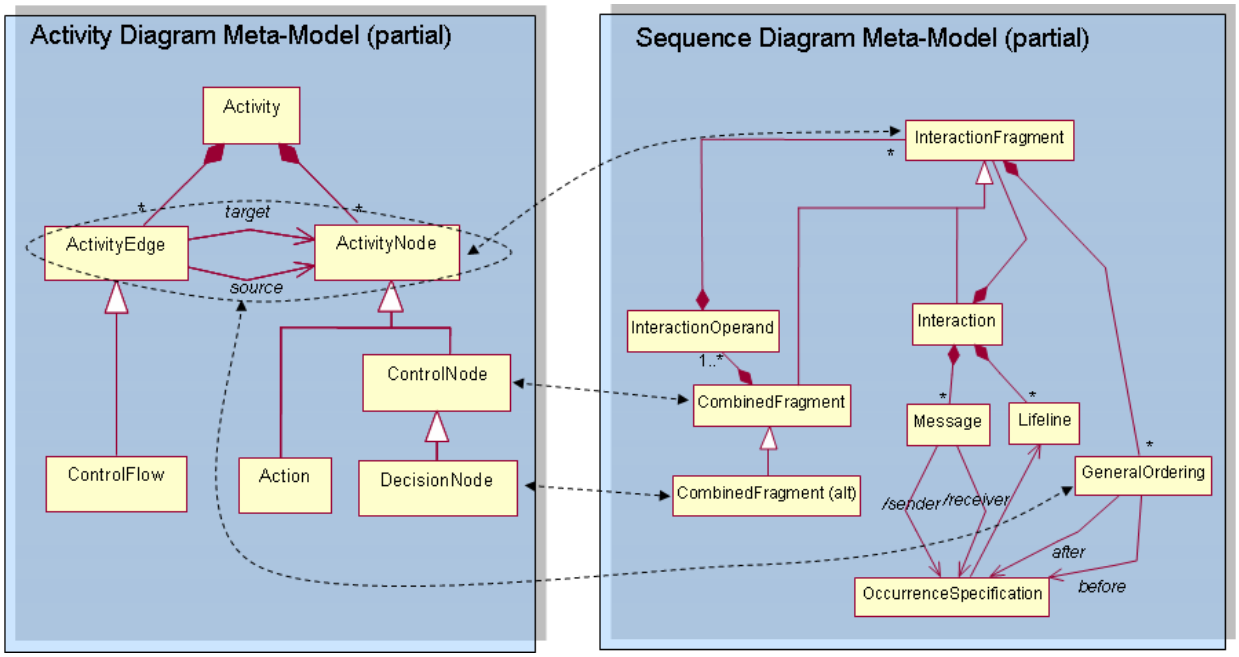

Figure 14. Meta-models of activity diagram and sequence diagram.

Given the elements in the meta-models along with their relationships, and their counterparts in our model, we can then present the algorithm (Figure 13) in terms of the meta-models and their relationships. For example, an activity diagram *AD* is an instance of activity diagram meta-model. Each of the state in *AD* is converted to an instance of `ActivityNode`, and a transition Δ in *AD* is a tuple $\tilde{\Delta}$ formed from instances of `ActivityEdge` and `ActivityNode`. A sequence diagram *SD* is an instance of sequence diagram meta-model. A region in *SD* is an instance of `InteractionFragment`, and the ordering of regions is specified

in terms of `GeneralOrdering`. The relations ρ between states and regions are given by the relationships $\tilde{\rho}$ between instances of `ActivityNode` and `InteractionFragement`. The relations ρ_{AD} between the states in two activity diagrams can be converted to an association $\tilde{\rho}_{AD}$ that relates instances of `ActivityNode` between two instances of activity diagram meta-model, which represent the two different activity diagrams. Instead of finding the difference of states (*addedStates* and *removedStates*) between AD_1 and AD_2, the algorithm finds the difference of nodes (*addedNodes* and *removedNodes*) between the corresponding meta-model counterparts of AD_1 and AD_2. To facilitate that, the *NotCovered* function (Definition 9) can be slightly modified such that it operates on instances of meta-model counterparts. Initialization of meta-model counterparts of SD_2 and ρ_2 is straightforward. By utilizing $\tilde{\Delta}_2$ (transition tuple of activity diagram meta-model version 2), $\tilde{\rho}_1$ (meta-model counterpart of ρ_1), and $\tilde{\rho}_{AD}$, we can find a placeholder for meta-model counterpart of a new region for every new element in *addedNodes*, and update $\tilde{\rho}_2$ (meta-model counterpart of ρ_2). Finally, we can check against order violations as in Figure 13 using the corresponding meta-model counterparts.

As we mentioned, we map a state in an activity diagram to an instance of `ActivityNode` in a meta-model. Technically speaking, we cannot do such mapping because `ActivityNode` is an abstract class that cannot be instantiated. This is one of the examples that there is no direct mapping relationship between our model and the meta-model elements. Another example is `InteractionFragment` in the meta-model which represents an interaction or a fragment in a sequence diagram. When we map a region to an `InteractionFragment` instance, we intend to map it to a fragment but not an interaction. To fix the first problem, we

can blindly instantiate `Action`, and this should not break our algorithm as discussed in Section 4.4. Alternatively, we instantiate either `Action` or `DecisionNode` provided that we have sufficient information to decide (see Section 4.4). To fix the second problem, we can use constraints to strengthen the types of meta-model elements so that we control what metal-model elements gets mapped to our model elements. While it is somewhat more difficult, we can still leverage meta-model based tools to implement our algorithm.

6 Conclusion and Future Work

The process of manually modifying software to meet its changing requirements is expensive and error-prone. Our goal is to provide assistance to the users for automatically propagating changes across software artifacts. In this thesis, activity and sequence diagrams are used to represent models at the requirements and design levels respectively, and describe a software system from different perspectives. We gave a notion of regions in sequence diagrams to give an abstraction of sequence of messages such that it closes the gap between model elements from different levels of abstraction. We formalized the relationships between activity and sequence diagrams, and provided conditions to validate these relationships. For example, one of such well-formedness rules was checking that the ordering of activities and regions correspond to each other. In addition, we described how our algorithm utilizes the inter-model relationships to automatically propagate changes across these models, and locate potential regions in sequence diagrams that require manual fix. Further, we discussed how the algorithm could be implemented in existing modeling tools.

Our algorithm thus far is demonstrated on activity diagrams with if-decision controls, and it assumes that every state has a single predecessor, but we also discussed how the algorithm can be extended to work for loop controls, i.e., cases where a state has more than one predecessor. Moreover, we explained why our algorithm works without distinguishing between decision nodes and action nodes, and discussed how to add typed states to the algorithm for deciding the detailed locations of new regions.

Of course, the work is far from being complete. Specifically, so far we have not looked at relating models other than activity and sequence diagrams. Our initial investigation into relating these models with Java code (or other implementation models) only indicated how challenging the problem is.

Change propagation between activity diagrams and sequence diagrams described in this thesis is part of our ongoing work to enable semi-automated change propagation in the MDD setting. The algorithm outlined in Chapter 4 is being implemented on top of our MMTF [13] framework, and we are planning to conduct additional case studies to understand what other relationships and rules need to be specified to propagate changes as models are evolving. We also expect to identify domain-specific rules, on the model and the meta-model levels, which would help construct meaningful relationships.

We have also showed that fully automating change propagation on models constructed at different levels of abstraction is impossible (Section 4.4), and that our process results in models with “unknowns” that require designer interaction. Formalizing this notion and enabling reasoning about it, as well as proofs of correctness of our approach are again left for future work.

References

[1] IBM Trade6 Benchmark, http://www.ibm.com/developerworks/edu/dm-dw-dm-0506lau.html, Jun 2005.

[2] J. Kramer and J. Magee. “Self-Managed Systems: an Architectural Challenge”. In Future of Software Engineering, pages 259–268, 2007.

[3] J. Zhang and B. Cheng. “Model-Based development of Dynamically Adaptive Software”. In Proceedings of International Conference on Software Engineering (ICSE’06), pages 371–380, 2006.

[4] L. Briand, Y. Labiche, L. O’Sullivan, and M. S´owka. “Automated Impact Analysis of UML Models”. Journal of Systems and Software, 79(3):339–352, 2006.

[5] A. Maule, W. Emmerich, and D. Rosenblum. “Impact Analysis of Database Schema Changes”. In Proceedings of International Conference on Software Engineering (ICSE’08), pages 451–460, 2008.

[6] A. Egyed, E. Letier, and A. Finkelstein. “Generating and Evaluating Choices for Fixing Inconsistencies in UML Design Models”. In Proceedings of International Conference on Automated Software Engineering (ASE’08), pages 99–108, 2008.

[7] M. Sabetzadeh, S. Nejati, S. Liaskos, S. Easterbrook, and M. Chechik. “Consistency Checking of Conceptual Models via Model Merging”. In Proceedings of Requirements Engineering Conference (RE ’07), pages 221–230, 2007.

[8] S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, and P. Zave. “Matching and Merging of Statecharts Specifications”. In Proceedings of International Conference on Software Engineering (ICSE ’07), pages 54–64, 2007.

[9] A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. “Inconsistency handling in multiperspective specifications”. IEEE Transactions on Software Engineering, 20(8):569–578, 1994.

[10] C. Nentwich, W. Emmerich, A. Finkelstein, and E. Ellmer. “Flexible consistency checking”. ACM Transactions on Software Engineering and Methodology, 12(1):28–63, 2003.

[11] G. Chintalapani. “Online Stock Brokerage System”. http://www.isr.umd.edu/~austin/ense621.d/projects04.d/project_gouthami.html, Fall 2003.

[12] Object Management Group. “OMG Unified Modeling Language (OMG UML) Superstucture, v 2.1.2”. http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF, Feb 2007.

[13] R. Salay, M. Chechik, S. Easterbrook, Z. Diskin, P. McCormick, S. Nejati, M. Sabetzadeh, and P. Viriyakattiyaporn. “An Eclipse-Based Tool Framework for Software Model Management”. In Proceedings of the 2007 OOPSLA workshop on Eclipse Technology eXchange, pages 55-59, October 2007.