Relationship-Based Change Propagation: A Case Study

Marsha Chechik Winnie Lai Shiva Nejati Jordi Cabot Zinovy Diskin Steve Easterbrook
Mehrdad Sabetzadeh Rick Salay
Department of Computer Science
University of Toronto, Toronto, ON, Canada.
{chechik,wlai,shiva,jcabot,zdiskin,sme,mehrdad,rsalay}@cs.toronto.edu

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 and error-prone. Thus, it is essential to assist users in propagating changes across requirements, design, and implementation artifacts.

In this paper, 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 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 Trade 6 case study to demonstrate our approach.

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* [5, 13], *software evolution* [1], and *change impact analysis* [7, 3]. 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.

In our earlier work, we have studied relationships between homogeneous models, i.e., models defined in the same notation. In particular, we have characterized syntactic and semantic relationships between structural models, such as class diagrams and ER diagrams [10], and behavioural models, such as state machines [8]. Further, we have developed semi-automated algorithms for computing such relationships [8]. Here, we build on our 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 metamodels.

In this paper, we describe our model-based change propagation technique by demonstrating it on a case study: an IBM WebSphere Performance Benchmark Sample called Trade 6 (see Section 2). Specifically, we show how relationships between heterogeneous models can be defined, and how the validity of these relationships can be checked using metamodel-level mappings (Section 3). We also 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 (Sections 4-5). Section 6 compares our work with the related research, and Section 7 concludes the paper with a discussion of future research directions.

2 Example

We motivate our work using Trade 6 [12] – an example of an online brokerage application, designed for benchmarking web service performance. Using Java and IBM WebSphere packages, it implements standard use cases for online trading: getting account profile, getting a stock quote, buy order, and sell order.

This example was chosen by our funding partner but, while being a well designed system, its documentation does not include explicit requirements and design models.

To acquire those, we realized that Trade 6 implements a relatively standard online brokerage system, like those available on the web [2]. Thus, we chose a use case, buy order, and obtained an activity diagram (AD) for placing an order from [2][1]. This diagram is shown on the left hand side of Figure 1. For this paper, we refer to this AD as the requirements model for this use case. In this AD, the user has already been logged into the system. He/She begins by entering the stock name and the number of stocks to buy. The order is then placed on the queue for processing. Finally, when the oder is finished, the system notifies the user.

We have further obtained a sequence diagram (SD) of this use case by (manually) reverse-engineering source code implementing it. This SD appears on the right hand side of Figure 1, and we refer to it as the design model for the buy order use case. A participant of type `TradeAction`, which represents the user, sends a `buy` message to `TradeServices`. The later processes the message by communicating the retrieval information of the user account and the stock quote to `DB`. Afterwards, it sends a `queueOrder` message to `TradeBrokerMDB`, which represents a trade exchange. After `TradeBrokerMDB` completes the order, it sends a message back to `TradeServices` which, in turn, updates the user account and the status of the order with `DB`, and sends an `orderCompleted` message to `TradeAction`.

3 Specifying Relationships

As we mention in Section 1, effective specification of relationships between models is at the center of our method. However, in this paper, we just illustrate relationships on our example (which were constructed by hand) rather than discussing how to relate models in general.

The relationships between states of AD and messages of SD in our example are shown as dashed lines in Figure 1. We note that an activity can be mapped to a single message or a sequence of messages (though it may not happen that the same message is part of the mapping of two different activities) (Rule R_1). Intuitively, this is because an SD represents a design-level model which is more defined than a requirements model represented by an AD. For example, the activity labeled s_1 in AD is mapped to a single message `buy` in SD, whereas the activity s_2 is mapped to a sequence of messages

$$< \texttt{getAccountData}, \texttt{getQuoteData}, \texttt{createOrder}, \texttt{queueOrder} >$$

We refer to a (one or sequence of) messages as a *region*. The order of the activities in the AD should match the order if the (sequence of) messages in the corresponding SD (Rule R_2). This is due to the fact that both diagrams describe the same behavior model of the system, although at different levels of abstraction. For example, the activity s_1 in the AD is followed by s_2, and the corresponding `buy` message is followed by the corresponding sequence of messages.

Rules R_1 and R_2 described above indicate some of the well-foundness rules of the relationship between ADs and SDs. First, we connect elements on the metamodel level of the corresponding diagrams (see Figure 2): activities and interaction fragments, and connections between control-flow elements. The actual rules are then formalized in OCL. As an example, rule R_1 could be defined as:

context Message **inv** R1:
$self.interaction.activityNode{-}{>}size() \leq 1$

Clearly, such descriptions land themselves to natural implementations of relationship checking within model management tools, such as our own tool MMTF [11].

4 Change Propagation

In this section, we illustrate how relationships defined in Section 3 are used to help propagate changes made to system requirements.

We now make a change requested by potential stakeholders of the Trade 6 system. Specifically, we enhance the buy order use case to ensure that the order is filled within a specified time frame. The new activity diagram is shown in Figure 3. The new requirement is captured as an addition of a condition c_1 to the original AD of the buy order. If the order is within the time limit, it is executed, and the system sends

---

[1] The original activity diagram includes the flow for both buy and sell orders, but we use it to describe buy orders only.

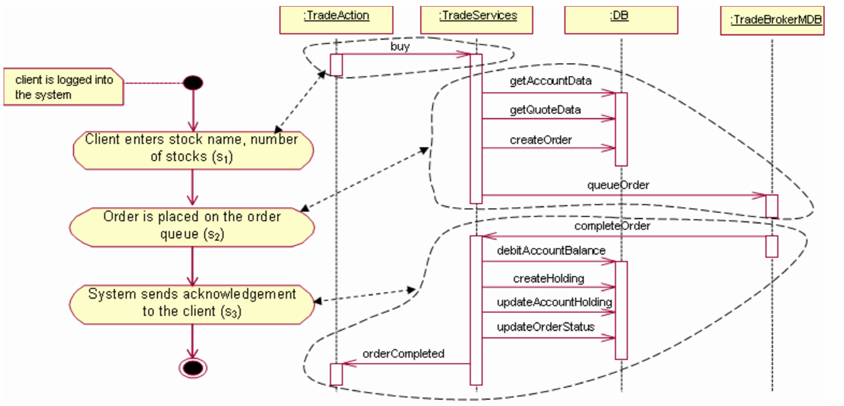

Figure 1. Buy order scenario: Relating states of the AD (LHS) to messages in the SD (RHS).

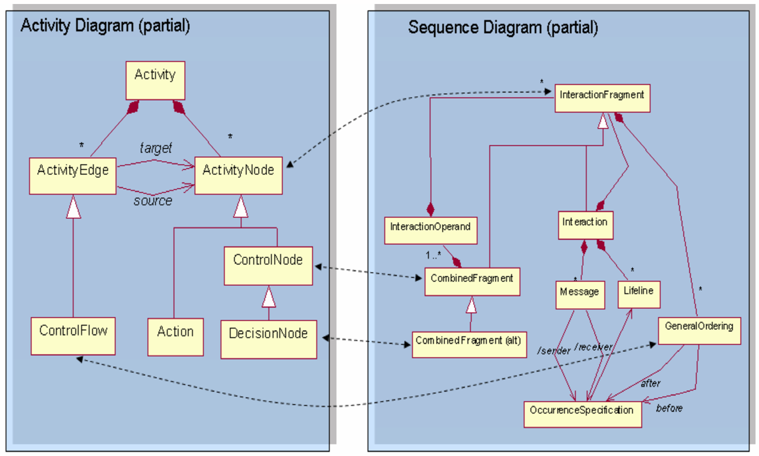

Figure 2. Fragments of AD and SD metamodels, and relationships between their elements.

an acknowledgement to the client (s_3); otherwise, it sends an `order expired` message.

As requirements change, the corresponding designs need to evolve accordingly, and maintaining the consistency between the different models is a major undertaking. We propose to automate propagation of changes to the related models. In what follows, we first show how a desired SD reflecting the above change should look like and then discuss the algorithm which can create this changed model semi-automatically.

The SD for the enhanced buy order use case appears in Figure 4. It reflects the corresponding changes in the AD in Figure 3. Specifically, it includes a new combined alternate interaction fragment. If the `within time` $-$ `limit` constraint is satisfied (see the upper fragment), the `completeOrder` message is sent, and the `TradeServices` participant proceeds as in the original SD. If the constraint is not met (the lower fragment), an `orderExpired` message is sent and then propagated to the `TradeAction` participant.

We now show how the relationships between requirements and design models established in Section 3 can be used to help users evolve the design of the SD for buy order, in response to changes in requirements. This idea is illustrated in Figure 5. The left-hand side represents the original buy order use case, with its activity diagram (we call it AD_1) in the top left and its sequence diagram (we call it SD_1) in the lower left. The relationships between AD_1 and SD_1 are those captured in Figure 1. The right hand side represents the enhanced buy order use case mentioned above. The top right diagram is its activity diagram (AD_2), also shown in Figure 3. The lower right diagram is the corresponding SD, referred to as SD_2 and also appearing in Figure 4. Since AD_2 is evolved from the original AD AD_1, we can distinguish between activities common to both diagrams and those new in AD_2. In particular, the precondition and the first two activities, s_1 and s_2, are present in both diagrams. The activ-

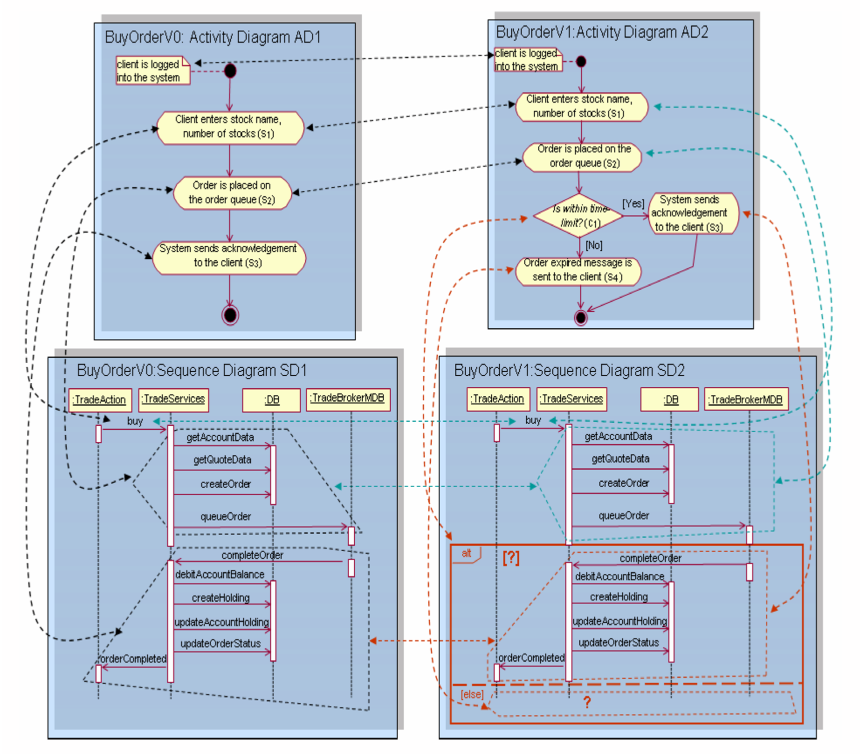

Figure 5. Using relations (dashed lines) to propagate changes.

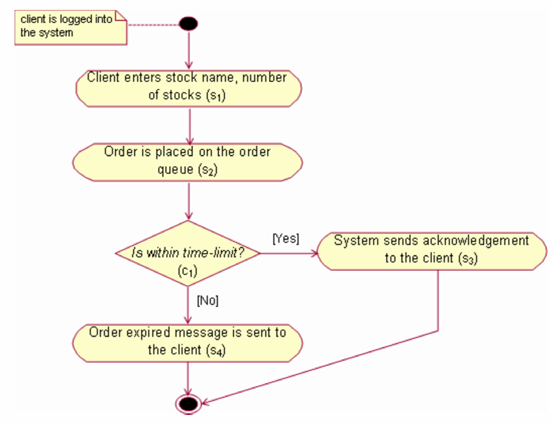

Figure 3. AD of the enhanced buy order.

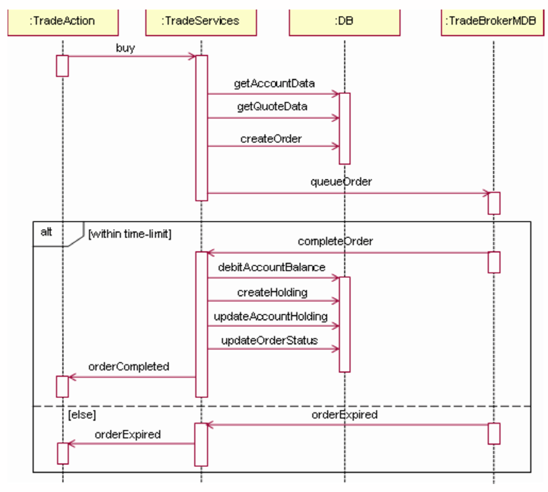

Figure 4. SD of the enhanced buy order.

ity `System sends acknowledgement to the clients` (s_3) is also present but has different preconditions in the two diagrams. AD_2 also has an additional condition (`Is within time` $-$ `limit`, c_1) and a new activity (`order expired message is sent to the client`, s_4), added to one of the branches of the condition check.

Intuitively, SD regions in the original diagram corresponding to the unchanged activities should be preserved in the new diagram. For example, the first two regions of SD_1 ($<$ `buy` $>$, and $<$ `getAccountData`, `getQuoteData`, `createOrder`, `queueOrder` $>$) are preserved in SD_2. SD regions in the original diagram which correspond to the activities which changed their precondition should appear in the new diagram as well (and in the original order), but their location may be different. For example, the SD_1 message sequence $<$ `completeOrder`, `debitAccountBalance`, `createHolding`, `updateAccountHolding`, `updateOrderStatus`, `orderCompleted` $>$ corresponds to the changed activity s_3 in AD_1. Thus, this sequence needs to appear in SD_2 but in a different location. Finally, the additions to AD_2, the new condition check c_1 and the new activity s_4 should be appropriately reflected in SD_2, with an addition of an `alt` operator and a new `else` block.

5 Automating Change Propagation

We now give high-level pseudocode for an algorithm for automating the relationship-based change propagation exemplified in Section 4. A more formal version of this algorithm can be found in [6].

Suppose we are given a version of an activity diagram AD_1 and its corresponding sequence diagram SD_1. Let S_1 be the states in AD_1, and RE_1 be the regions in SD_1 and assume that the relationship between RE_1 and S_1, called ρ_1 is available as well. In addition, we are given a new version of activity diagram, AD_2 (with states S_1) and a relationship ρ_{AD} that relates S_1 and S_2. Our goal is to automatically compute changes needed to be made in the new sequence diagram. We do so in an algorithm LOCATECHANGE, shown in Figure 6.

The algorithm starts by looking at the difference between states of AD_1 and AD_2, storing them in *addedStates* and *removedStates*. Then it initializes the new sequence diagram SD_2 by copying the regions from SD_1 whose corresponding states are not in *removedStates*. It also initializes the relation, ρ_2, between the regions in SD_2 and the states S_2 in AD_2. This is done by (1) taking the regions of SD_2, copied from SD_1; (2) finding the corresponding states in AD_1 via the relation ρ_1; and (3) using the relation ρ_{AD} to find the states in AD_2 mapped to states of AD_1 identified in the previous step.

After these initializations, the algorithm iterates over every state in *addedStates* to produce placeholders for regions of 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. This placeholder indicates the location of the new region sd in SD_2 that corresponds to the new state y. The relationship ρ_2 is also updated with the relation between sd and y. If a 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 add a new region for y in SD_2; instead we extend the placeholder of the region for x in SD_2 to also hold the messages corresponding to y. This is a design decision made to minimize the number of placeholders in the resulting SDs.

Finally, the algorithm checks for potential violations of the ordering constraint (rule R_2) 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 look for the region sd' of y and the region sd of x in SD_2 by using ρ_2, and then check whether the ordering between x and y is in conflict with the ordering between sd and sd'.

```
Algorithm. LOCATECHANGE
Input:  $AD_1$: An AD, version 1.
        $AD_2$: An AD, version 2.
        $SD_1$: An SD, version 1, corresponding to $AD_1$.
        $\rho_1$:   A relation between $SD_1$ and $AD_1$.
        $\rho_{AD}$: A relation between $AD_1$ and $AD_2$.
Output: $SD_2$: An SD version 2, corresponding to $AD_2$.
        $\rho_2$:   A relation between $SD_2$ and $AD_2$.

Let addedStates be states in $AD_2$ but whose corresponding
  (via $\rho_{AD}$) states are not in $AD_1$
Let removedStates be states in $AD_1$ but whose corresponding
  (via $\rho_{AD}$) states are not in $AD_2$
Initialize $SD_2$ with $SD_1$, but only keep regions of $SD_1$
  whose corresponding (via $\rho_1$) states are not in removedStates
Initialize $\rho_2$ by copying those tuples $(x, y)$ of $\rho_1$ such that
  $x$ is not in removedStates
For every state $s$ in addedStates
    Insert a placeholder region $r$ corresponding to $s$ in $SD_2$
    Update $\rho_2$ to include $(s, r)$
    Check if $\rho_2$ is a valid relation between $AD_2$ and $SD_2$
      using well-foundness rules (see Section 3)
    Report any violations caused by $\rho_2$
```

Figure 6. Algorithm for locating changes.

In the example in Figure 5, *addedStates* is $\{c_1, s_4\}$, and *removedStates* is an empty set; thus $\{s_1, s_2, s_3\}$ 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. The predecessor of the new state s_4 in AD_2 is c_1. Since c_1 is not found in the relation ρ_{AD}, we assume that the placeholder for the region of c_1 in SD_2 should be extended to hold the messages of s_4. After handling the new states, we check AD_2 against order violations (rule R_2). 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.

6 Related Work

Specifying relationships between a set of heterogeneous models has been previously studied: [4] 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. [9] 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.

Our work is most closely related to the efforts on *impact analysis* [3] and *change propagation* in the context of software engineering models[1]. [3] uses consistency rules to determine, as the change is made, which of the instances need to be reevaluated. [1] 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.

7 Conclusion and Discussion

Change propagation between activity diagrams and sequence diagrams described in this paper is part of our ongoing work to enable semi-automated change propagation in the MDD setting. The algorithm outlined in Section 5 is being implemented on top of our MTTF [11] framework, and we are planning to do additional case studies to understand what other relationships and rules need to be specified to propagate changes as models are being evolved. We also expect to identify domain-specific rules, on the model and the meta-model levels, which would help construct meaningful relationships.

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

We have also showed that fully automating change propagation on models constructed at different levels of abstraction is impossible, 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.

Acknowledgments. We are grateful to members of the CERAS project and to Richard Paige for many interesting discussions.

References

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

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

[3] A. Egyed, E. Letier, and A. Finkelstein. “Generating and Evaluating Choices for Fixing Inconsistencies in UML Design Models”. In *Proceedings of ASE’08*, pages 99–108, 2008.

[4] A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency handling in multiperspective specifications. *IEEE TSE*, 20(8):569–578, 1994.

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

[6] W. Lai. “Towards a Relationship-Based Change Propagation”. Master’s thesis, University of Toronto, Department of Computer Science, February 2009.

[7] A. Maule, W. Emmerich, and D. Rosenblum. “Impact Analysis of Database Schema Changes”. In *Proceedings of ICSE’08*, pages 451–460, 2008.

[8] S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, and P. Zave. “Matching and Merging of Statecharts Specifications”. In *Proceedings of ICSE ’07*, pages 54–64, 2007.

[9] C. Nentwich, W. Emmerich, A. Finkelstein, and E. Ellmer. Flexible consistency checking. *ACM TOSEM*, 12(1):28–63, 2003.

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

[11] 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 ETX’07 at OOPSLA’07*, October 2007.

[12] IBM Trade6 Benchmark, June 2005. Available at: `http://www.ibm.com/developerworks/edu/dm-dw-dm-0506lau.html`.

[13] J. Zhang and B. Cheng. “Model-Based development of Dynamically Adaptive Software”. In *Proceedings of ICSE’06*, pages 371–380, 2006.